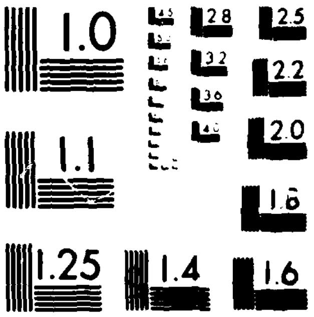
AD-A165 076   ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA   1/7
TOPICS L305 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K506

UNCLASSIFIED   F/G 9/2   NL



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

AD-A165 076 1 1986

Ada® Training Curriculum

Advanced Ada® Topics
L305
Teacher's Guide
Volume II

Supersedes AD A144 799

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K506

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

*Approved For Public Release/Distribution Unlimited

PART IV. OTHER ABSTRACTION FEATURES

10. OVERLOADING

11. GENERIC UNITS

12. DERIVED TYPES

13. UNCHECKED DEALLOCATION

INSTRUCTOR NOTES

SECTION 10

OVERLOADING

INSTRUCTOR NOTES

OVERVIEW.

ENTRIES (DISCUSSED LATER IN VIII.D) CAN ALSO BE OVERLOADED.

OVERLOADING IS ONLY ALLOWED FOR ENUMERATION LITERALS, SUBPROGRAMS, AND ENTRIES.

OVERLOADING

AN IDENTIFIER CAN BE DECLARED TO HAVE MORE THAN ONE MEANING IN A GIVEN PART OF THE PROGRAM.

THE CONTEXT OF A PARTICULAR USE OF AN OVERLOADED IDENTIFIER DETERMINES WHICH MEANING IS APPLICABLE.

OVERLOADING IS ALLOWED FOR IDENTIFIERS THAT ARE DECLARED AS ENUMERATION LITERALS OR AS SUBPROGRAMS.

INSTRUCTOR NOTES

TWO VERSIONS OF THE FUNCTION SIGN ARE DECLARED TOGETHER, CAUSING THEM TO BE OVERLOADED. ONE HAS A PARAMETER AND A RESULT OF TYPE Integer, THE OTHER A PARAMETER AND A RESULT OF TYPE Float. BECAUSE THESE FUNCTIONS DO ESSENTIALLY THE SAME THING ON TWO DIFFERENT TYPES (THE ONLY DIFFERENCE BEING THE USE OF Integer LITERALS VERSUS REAL LITERALS), IT IS APPROPRIATE TO GIVE THEM THE SAME NAME. OF COURSE THERE IS NO WAY TO ENFORCE THIS STYLE GUIDELINE. THE TWO FUNCTION BODIES COULD HAVE BEEN RADICALLY DIFFERENT FROM EACH OTHER.

BECAUSE THE FIRST CALL HAS A Float PARAMETER, AND BECAUSE THE RESULT IS ASSIGNED TO A Float VARIABLE, IT MUST BE A CALL ON THE VERSION OF Sign DECLARED FIRST. BY SIMILAR REASONING, THE SECOND CALL MUST BE A CALL ON THE VERSION OF Sign DECLARED SECOND.

THE OVERLOADING CREATES THE ILLUSION THAT THERE IS A SINGLE FUNCTION NAMED Sign THAT WORKS WITH EITHER Integer OR Float VALUES.

EXAMPLE OF OVERLOADING

```
procedure P is
   F, G : Float;
   I, J : Integer;
   function Sign (X : Float) return Float is
   begin
      if X < 0.0 then
         return -1.0;
      elsif X > 0.0 then
         return 1.0;
      else
         return 0.0;
      end if;
   end Sign;
   function Sign (X : Integer) return Integer is
   begin
      if X < 0 then
         return -1;
      elsif X > 0 then
         return 1;
      else
         return 0;
      end if;
   end Sign;
   -- THE NAME Sign IS OVERLOADED
begin -- P
   ...
   F := Sign (G); -- A CALL ON THE FIRST FUNCTION (FOR TYPE Float)
   ...
   I := Sign (J); -- A CALL ON THE SECOND FUNCTION (FOR TYPE Integer)

end P;
```

INSTRUCTOR NOTES

- DECLARATIONS:

  THESE VERSIONS OF Put ARE THE ONES OBTAINED BY INSTANTIATING Text_IO.Integer_IO WITH TYPE Integer AND Text_IO.Float_IO WITH TYPE Float. THUS CALLS ON Put WITH A SINGLE ARGUMENT OF TYPE Integer OR A SINGLE ARGUMENT OF TYPE Float ARE ALLOWED.

- CALLS:

  1. A RESULT OF TYPE Integer IS REQUIRED, SO THE CALL MUST BE ON VERSION 1.
  2. A RESULT OF TYPE Float IS REQUIRED, SO THE CALL MUST BE ON VERSION 2.
  3. Put MAY BE CALLED WITH A SINGLE ARGUMENT OF EITHER TYPE Integer OR TYPE Float.
  4. THE QUALIFIED EXPRESSION MUST CONTAIN AN EXPRESSION OF TYPE Integer, SO THE CALL MUST BE ON VERSION 1. QUALIFIED EXPRESSIONS ARE REVIEWED ON THE NEXT SLIDE.

AMBIGUITY

- USUALLY, THE CONTEXT IN WHICH AN OVERLOADED SUBPROGRAM IS CALLED IDENTIFIES THE VERSION BEING CALLED.

- IF THE CONTEXT IS NOT SUFFICIENT TO IDENTIFY A UNIQUE VERSION, THE CALL IS AMBIGUOUS.

- Ada COMPILERS REJECT AMBIGUOUS CALLS AS ERRORS.

- DECLARATIONS

```
function Ceiling (X : Float) return Integer; -- VERSION 1
function Ceiling (X : Float) return Float;   -- VERSION 2

procedure Put
   (Item  : in Integer;
    Width : in Field := Default_Width;
    Base  : in Number_Base := Default_Base); -- FROM Text_IO.Integer_IO

procedure Put
   (Item : in Float;
    Fore : in Field := Default_Fore;
    Aft  : in Field := Default_Aft;
    Exp  : in Field := Default_Exp); -- FROM Text_IO.Float_IO

I    : Integer;
F, G : Float;
```

- CALLS

```
I := Ceiling (G);                -- CALL ON VERSION 1
F := Ceiling (G);                -- CALL ON VERSION 2
Put (Ceiling (G));               -- AMBIGUOUS, THEREFORE ILLEGAL
Put (Integer'(Ceiling (G)));     -- CALL ON VERSION 1 (QUALIFIED EXPRESSION)
```

INSTRUCTOR NOTES

THIS FOIL IS SELF EXPLANATORY.

OVERLOADING OF SUBPROGRAMS

- THE SAME IDENTIFIER MAY BE DECLARED AS THE NAME OF MORE THAN ONE SUBPROGRAM, EVEN WITHIN THE SAME SEQUENCE OF DECLARATIONS.
  - THE SUBPROGRAM DECLARATIONS MUST HAVE DISTINCT FORMAL PARAMETER AND RESULT TYPE COMBINATIONS.
  - ALL OF THE SUBPROGRAM DECLARATIONS ARE SIMULTANEOUSLY VISIBLE.

- PARTICULARLY USEFUL WHEN YOU WANT TO APPLY SIMILAR ACTIONS TO DIFFERENT TYPES.

- EXAMPLE:
  - THE PREDEFINED LIBRARY PACKAGE Text_IO DECLARES OVERLOADED SUBPROGRAMS Get AND Put FOR TYPES Character, String, AND FOR Integer, Float, Fixed, AND Enumeration TYPES.
  - THE FOLLOWING CALLS TO THE VARIOUS Put SUBPROGRAMS HAVE THE SIMILAR EFFECT OF CONVERTING THE ARGUMENT VALUE TO A STRING AND WRITING THE STRING TO THE CURRENT OUTPUT FILE.

```
Put ('A');
Put ("ABC");
Put (12);
Put (12.3);
Put (True);
```

INSTRUCTOR NOTES

THESE EXAMPLES ILLUSTRATE WHY OVERLOADING OF ENUMERATION LITERALS IS DESIRABLE. FOR EXAMPLE, IN THE REAL WORLD, Low AND High MAY BE BOTH SECURITY LEVELS AND FAN SETTINGS, AND THESE FACTS ARE DIRECTLY (AND MNEMONICALLY) EXPRESSIBLE IN THE Ada CODE.

OVERLOADING OF ENUMERATION LITERALS

- THE SAME IDENTIFIER MAY BE USED AS AN ENUMERATION LITERAL FOR MORE THAN ONE TYPE:

```
type Fan_Setting_Type is (Off, Low, Medium, High);
type Lamp_Setting_Type is (On, Off);
type Security_Level_Type is (High, Low);
```

- ENUMERATION VALUES OF DIFFERENT TYPES ARE DISTINCT VALUES, EVEN IF THEY HAPPEN TO HAVE THE SAME NAME. THEY HAVE NO SPECIAL RELATIONSHIP.

  - Fan_Setting_Type'Val (0) /= Lamp_Setting_Type'Val (1), EVEN THOUGH BOTH ARE NAMED OFF
  - THE Fan_Setting_Type VALUE Low is LESS THAN THE Fan_Setting_Type VALUE High, BUT THE Security_Level_Type VALUE Low is GREATER THAN THE Security Level_Type VALUE High

INSTRUCTOR NOTES

PARAMETER AND RESULT TYPE PROFILES

- SUBPROGRAMS MAY ONLY BE OVERLOADED IF THEY HAVE DIFFERENT PARAMETER AND RESULT TYPE PROFILES.

- A PARAMETER AND RESULT TYPE PROFILE DESCRIBES THE BASE TYPES OF A SUBPROGRAM'S PARAMETERS (IN ORDER) AND RESULT. (FOR PROCEDURES, THE RESULT TYPE IS "NONE.")

- EXAMPLES:

```
function "**" (Left : Integer; Right : Natural) return Integer;

     PROFILE : | PARAMETER BASE TYPES : (1) Integer; (2) Integer
               | RESULT BASE TYPE     : Integer

procedure P (X1 : in Character := 'X'; X2 : in out Natural; X3 : out Positive);

     PROFILE : | PARAMETER BASE TYPES : (1) Character; (2) Integer; (3) Integer
               | RESULT BASE TYPE     : NONE

function Clock return Time;

     PROFILE : | PARAMETER BASE TYPES : NONE
               | RESULT BASE TYPE     : Time

procedure Clear_Screen;

     PROFILE : | PARAMETER BASE TYPES : NONE
               | RESULT BASE TYPE     : NONE
```

INSTRUCTOR NOTES

OBSERVATIONS ABOUT PARAMETER AND RESULT TYPE PROFILES

- FUNCTIONS AND PROCEDURES NEVER HAVE THE SAME PROFILE
- SUBPROGRAMS WITH DIFFERENT NUMBERS OF FORMAL PARAMETERS NEVER HAVE THE SAME PROFILE
- PROFILES DO NOT REFLECT ANY OF THE FOLLOWING:
  - FORMAL PARAMETER NAMES
  - PARAMETER MODES (in, in out, out)
  - EXISTENCE OR CONTENT OF DEFAULT PARAMETER VALUE EXPRESSIONS
  - PARAMETER SUBTYPES (ONLY BASE TYPES ARE REFLECTED)

```
function S (A : Positive; B : Integer := 0) return Natural;
    PROFILE : PARAMETER BASE TYPES : (1) Integer; (2) Integer
              RESULT BASE TYPE     : Integer
    (ONLY THE BASE TYPE Integer APPEARS IN THE PROFILE, NOT THE SUBTYPES Positive AND
     Natural.  THE DEFAULT VALUE 0 AND THE PARAMETER NAMES A AND B ARE NOT REFLECTED.)
procedure S (A : in Integer; B : out Natural);
    PROFILE : PARAMETER BASE TYPES : (1) Integer; (2) Integer
              RESULT BASE TYPE     : NONE
    (PARAMETER MODES ARE NOT REFLECTED IN THE PROFILE.)
procedure S (C : in out Positive; D : in out Positive);
    PROFILE : PARAMETER BASE TYPES : (1) Integer; (2) Integer
              RESULT BASE TYPE     : NONE
    (DESPITE DIFFERENT PARAMETER NAMES, MODES, AND SUBTYPES, THIS VERSION OF S HAS THE
    SAME PROFILE AS THE PREVIOUS ONE, SO THESE VERSIONS MAY NOT BE OVERLOADED WITH EACH
    OTHER.)
procedure S (E : out Natural; F : in Integer);
    PROFILE : PARAMETER BASE TYPES : (1) Integer; (2) Integer
              RESULT BASE TYPE     : NONE
    (SAME REMARK AS ABOVE)
procedure S (G : in Natural := 1; H : in Positive := 2);
    PROFILE : PARAMETER BASE TYPES :
              RESULT BASE TYPE     :
    (DESPITE THE EXISTENCE OF DEFAULT PARAMETER VALUES, THE PROFILE IS THE SAME EVEN IF
    IT IS POSSIBLE TO CALL THIS VERSION OF S WITH 0 OR 1 PARAMETERS, OVERLOADING WITH
    ANY OF THE PREVIOUS THREE VERSIONS IS DISALLOWED.
function S return Natural;
    PROFILE : PARAMETER BASE TYPES : NONE
              RESULT BASE TYPE     : Integer
procedure S;
    PROFILE : PARAMETER BASE TYPES : NONE
              RESULT BASE TYPE     : NONE
```

FILL IN THE PARAMETER AND RESULT TYPE PROFILES

```
function S (A : Positive; B : Integer := 0) return Natural;

     PROFILE : | PARAMETER BASE TYPES :
               | RESULT BASE TYPE     :

procedure S (A : in Integer; B : out Natural);

     PROFILE : | PARAMETER BASE TYPES :
               | RESULT BASE TYPE     :

procedure S (C : in out Positive; D : in out Positive);

     PROFILE : | PARAMETER BASE TYPES :
               | RESULT BASE TYPE     :

procedure S (E : out Natural; F : in Integer);

     PROFILE : | PARAMETER BASE TYPES :
               | RESULT BASE TYPE     :

procedure S (G : in Natural := 1; H : in Positive := 2);

     PROFILE : | PARAMETER BASE TYPES :
               | RESULT BASE TYPE     :

function S return Natural;

     PROFILE : | PARAMETER BASE TYPES :
               | RESULT BASE TYPE     :

procedure S;

     PROFILE : | PARAMETER BASE TYPES :
               | RESULT BASE TYPE     :
```

INSTRUCTOR NOTES

THE RIGHTHAND SIDE OF THE ASSIGNMENT SETTING := Off; CAN BE VIEWED AS A FUNCTION CALL. CONSEQUENCES ARE EXPLAINED ON THE NEXT SLIDE.

EMPHASIZE THAT THIS SLIDE SAYS NOTHING ABOUT THE ACTUAL IMPLEMENTATION OF ENUMERATION VALUES. (TYPICALLY, THEY ARE ENCODED AS INTEGERS, AND EVALUATION OF AN ENUMERATION LITERAL DOES NOT INVOLVE THE OVERHEAD OF A FUNCTION CALL.)

IT SIMPLY SAYS THAT ENUMERATION LITERALS CAN BE VIEWED AS FUNCTION CALLS WHEN APPLYING OVERLOADING RULES. THIS ALLOWS A SINGLE SET OF RULES TO GOVERN BOTH SUBPROGRAMS AND ENUMERATION LITERALS.

PARAMETER AND RESULT TYPE PROFILES OF ENUMERATION LITERALS

- THE OVERLOADING RULES TREAT ENUMERATION LITERALS AS PARAMETERLESS FUNCTIONS RETURNING VALUES OF THE APPROPRIATE ENUMERATION TYPE:

```
type Fan_Setting_Type is (Off, Low, Medium, High);
-- TREATED BY OVERLOADING RULES AS:
      function Off return Fan_Setting_Type is
      begin
         return Fan_Setting_Type'Val (0);
      end Off;
      function Low return Fan_Setting_Type is
      begin
         return Fan_Setting_Type'Val (1);
      end Low;
      function Medium return Fan_Setting_Type is
      begin
         return Fan_Setting_Type'Val (2);
      end Medium;
      function High return Fan_Setting_Type is
      begin
         return Fan_Setting_Type'Val (3);
      end High;
```

```
PROFILE :
PARAMETER BASE TYPES : NONE
RESULT BASE TYPE     : Fan_Setting_Type
```

```
 type Security_Level_Type is (High, Low);
-- TREATED BY OVERLOADING RULES AS:
      function High return Security_Level_Type is
      begin
         return Security_Level_Type'Val (0);
      end High;
      function Low return Security_Level_Type is
      begin
         return Security_Level_Type'Val (1);
      end Low;
```

```
PROFILE :
PARAMETER BASE TYPES : NONE
RESULT BASE TYPE     : Security_Level_Type
```

INSTRUCTOR NOTES

CONSEQUENCES

- THE TWO VERSIONS OF High CAN BE OVERLOADED BECAUSE THEY HAVE DIFFERENT PROFILES. SIMILARLY FOR Low.

- ENUMERATION LITERALS IN DIFFERENT TYPES ALWAYS HAVE DIFFERENT PROFILES.

- PROFILES MAY RESTRICT THE OVERLOADING OF ENUMERATION LITERALS WITH FUNCTIONS:

```
type Fan_Setting_Type is (Off, Low, Medium, High);
procedure High is
begin
   Current_Setting := High;
   Move_Fan_Switch (To => Current_Setting);
end High;

function High return Boolean is
begin
   return Current_Setting = High;
end High;
function High return Fan_Setting_Type is
begin
   return Maximum_Setting_So_far;       -- **ILLEGAL
end High;
```

PROFILE : PARAMETER BASE TYPES : NONE
RESULT BASE TYPE : NONE
(OVERLOADING LEGAL)

PROFILE : PARAMETER BASE TYPES : NONE
RESULT BASE TYPE : Boolean
(OVERLOADING LEGAL)

PROFILE : PARAMETER BASE TYPES : NONE
RESULT BASE TYPE : Fan_Setting_Type
(OVERLOADING PROHIBITED)

INSTRUCTOR NOTES

ANSWERS:

2	Y							
3	N	N						
4	Y	Y	N					
5	N	N	N	N				
6	N	Y	N	Y	N			
7	Y	Y	N	Y	N	Y		
8	Y	N	N	Y	N	Y	Y	
9	Y	Y	N	Y	N	Y	Y	Y
	1	2	3	4	5	6	7	8

PRINCIPLES:

1. ONLY SUBPROGRAMS AND ENUMERATION LITERALS MAY BE OVERLOADED. [ALSO ENTRIES, BUT WE HAVEN'T COVERED THEM YET.]

2. TO BE OVERLOADED, TWO ENTITIES MUST HAVE DIFFERENT PROFILES.

3. IN DETERMINING PROFILES, ENUMERATION LITERALS ARE CONSIDERED TO BE PARAMETERLESS FUNCTIONS.

WHICH DECLARATIONS OF "A" CAN BE OVERLOADED WITH WHICH?

1. procedure $\boxed{A}$ (X : in Integer := 0);
2. type T1 is ($\boxed{A}$, B, C);
3. type A is array (Positive range<>) of Integer;
4. function $\boxed{A}$ (X : Integer) return T1;
5. $\boxed{A}$: Integer;
6. procedure $\boxed{A}$ (X : out Positive);
7. type T2 is ($\boxed{A}$, E, I, O, U);
8. function $\boxed{A}$ return T1;
9. procedure $\boxed{A}$;

2								
3								
4								
5								
6								
7								
8								
9								
	1	2	3	4	5	6	7	8

INSTRUCTOR NOTES

THIS IS A REVIEW OF SCOPE AND VISIBILITY RULES PRESENTED IN L202.

THE NEXT FEW SLIDES ADDRESS THE INTERACTION OF HIDING AND OVERLOADING.

ANOTHER MEANING OF HIDING

```
procedure Outer is
   T : Integer;
   procedure Inner is
      type T is range 0 .. 100; -- Hides outer declaration
   begin -- Inner
      T := 0; -- ILLEGAL.  OUTER T IS HIDDEN
      Outer.T := T'Last; -- ASSIGNS 100 TO Integer VARIABLE T.
   end Irner;
begin -- Outer
   T := 0: -- Legal
   T := Inner.T'Last; -- ILLEGAL.OUTSIDE SCOPE OF Inner.T
end Outer;
```

- *HOW DO HIDING AND OVERLOADING AFFECT EACH OTHER?*

INSTRUCTOR NOTES

TO PARAPHRASE BULLET 1, TWO DECLARATIONS CAN COEXIST IF AND ONLY IF THEY CAN BE OVERLOADED WITH EACH OTHER. THE EXERCISE ON THE PREVIOUS SLIDE REVIEWED THIS CONCEPT.

THE SECOND BULLET GIVES TWO SIMPLE RULES DEFINING THE RELATIONSHIP BETWEEN HIDING AND OVERLOADING. THE NEXT THREE SLIDES GIVE EXAMPLES.

WE SHALL USE THE NOTION OF DECLARATIONS THAT CAN COEXIST AGAIN IN A FEW SLIDES, WHEN DISCUSSING OVERLOADING AND use CLAUSES.

HIDING AND OVERLOADING

- TWO DECLARATIONS OF THE SAME NAME CAN COEXIST ONLY IF
  - EACH DECLARATION IS FOR A SUBPROGRAM OR ENUMERATION LITERAL
  - THE TWO DECLARATIONS HAVE DIFFERENT PARAMETER AND RESULT TYPE PROFILES
- IF TWO DECLARATIONS FOR THE SAME NAME OCCUR IN NESTED SCOPES, THEN WITHIN THE INNER SCOPE
  - THE TWO MEANINGS OF THE NAME ARE OVERLOADED IF THEIR DECLARATIONS CAN COEXIST
  - THE INNER MEANING HIDES THE OUTER MEANING IF THEIR DECLARATIONS CANNOT COEXIST

INSTRUCTOR NOTES

HIDING AND OVERLOADING -- EXAMPLE 1

```
declare

     procedure P (X : in out Natural);      -- P#1.  PROFILE : | PARAMETER BASE TYPES : (1) Integer
                                                               | RESULT BASE TYPE     : NONE

     procedure P (X : in out Float);        -- P#2.  PROFILE : | PARAMETER BASE TYPES : (1) Float
                                                               | RESULT BASE TYPE     : NONE

     I : Integer := 1;
     F : Float := 0.0;
     ...

begin

     declare
          P : Integer;                      -- P#3.
     begin
          P (I);    -- ILLEGAL; P#1 NOT VISIBLE.
          P (F);    -- ILLEGAL; P#2 NOT VISIBLE.
          P := I;   -- OK; USES P#3.
     end;

end;
```

INSTRUCTOR NOTES

HIDING AND OVERLOADING -- EXAMPLE 2

```
declare

     P : Integer;                                  -- P#1.

begin

     declare
          procedure P (X : in out Natural);        -- P#2.  PROFILE : | PARAMETER BASE TYPES : (1) Integer |
                                                                      | RESULT BASE TYPE     : NONE        |

          procedure P (X : in out Float);          -- P#3.  PROFILE : | PARAMETER BASE TYPES : (1) Float   |
                                                                      | RESULT BASE TYPE     : NONE        |

          I : Integer := 1;
          F : Float := 0.0;
          ...
     begin
          P := 2;   -- ILLEGAL; P#1 IS NOT VISIBLE.
          P (I);    -- INVOKES P#2.
          P (F);    -- INVOKES P#3.
     end;

end;
```

INSTRUCTOR NOTES

HIDING AND OVERLOADING -- EXAMPLE 3

```
declare
    procedure P (X1 : in Natural;  X2 : out Positive);          -- P#1.

        PROFILE : PARAMETER BASE TYPES : (1) Integer; (2) Integer
                  RESULT BASE TYPE     : NONE

    procedure P (X1 : in Positive; X2 : out Float);             -- P#2.

        PROFILE : PARAMETER BASE TYPES : (1) Integer; (2) Float
                  RESULT BASE TYPE     : NONE

    I : Integer := 1;
    F : Float := 0.0;
    ...

begin
    declare
        procedure P (Y1 : in Integer;  Y2 : out Natural);       -- P#3.
            -- HIDES P#1, OVERLOADS P#2.
        ...

        PROFILE : PARAMETER BASE TYPES : (1) Integer; (2) Integer
                  RESULT BASE TYPE     : NONE

    begin
        P (I, I);       -- INVOKES P#3.
        P (I, F);       -- INVOKES P#2.
    end;
    P (I, I);           -- INVOKES P#1.
end;
```

INSTRUCTOR NOTES

THE NEXT TWO SLIDES PROVIDE EXAMPLES.

OVERLOADING AND use CLAUSES

- NORMALLY, A use CLAUSE FOR PACKAGE P ALLOWS AN ENTITY X PROVIDED BY P TO BE REFERRED TO AS X INSTEAD OF P.X.

- CERTAIN RESTRICTIONS APPLY IF THERE IS ALREADY AN ENTITY NAMED X VISIBLE AT THE PLACE OF THE use CLAUSE

  - IF THE TWO DECLARATIONS OF X CAN COEXIST, THEY ARE OVERLOADED
  - IF THE TWO DECLARATIONS OF X CANNOT COEXIST, THE use CLAUSE DOES NOT APPLY TO P.X.

- SIMILAR RESTRICTIONS APPLY TO TWO use CLAUSES FOR PACKAGES THAT BOTH CONTAIN ENTITIES NAMED X:

  - IF THE TWO DECLARATIONS OF X CAN COEXIST, THEY ARE OVERLOADED
  - IF THE TWO DECLARATIONS OF X CANNOT COEXIST, THE use CLAUSE DOES NOT APPLY TO EITHER PACKAGE'S X

INSTRUCTOR NOTES

THE SITUATION WOULD HAVE BEEN EXACTLY THE SAME IF THE use CLAUSE HAD OCCURRED INSIDE THE DECLARATIVE PART OF Main.

INSIDE Main, THE SIMPLE NAMES A AND G MEAN Main.A AND Main.G.

OVERLOADING AND use CLAUSES -- EXAMPLE 1

```
package Pl is
   A : Integer;                                         -- A#1
   function F (X : Float) return Float;                 -- F#1
   function F (X : Integer) return Float;               -- F#2
   function G return Integer;                           -- G#1
end Pl;

with Pl; use Pl;

procedure Main is
   function A return Float;                             -- A#2
   function F (X : Integer) return Float;               -- F#3
   function F (X : Integer) return Integer;             -- F#4
   G : Integer;                                         -- G#2
begin -- Main
   -- A#1 MUST BE REFERRED TO AS Pl.A, SINCE IT CANNOT COEXIST WITH A#2.
   -- F#1 OVERLOADS F#3 AND F#4, AND MAY BE REFERRED TO AS F.
   -- F#2 MUST BE REFERRED TO AS Pl.F, SINCE IT CANNOT COEXIST WITH F#3.
   -- G#1 MUST BE REFERRED TO AS Pl.G, SINCE IT CANNOT COEXIST WITH G#2.
   ...
end Main;
```

INSTRUCTOR NOTES

THE SITUATION COULD HAVE EXACTLY THE SAME EFFECT IF use CLAUSES FOR ONE OR BOTH PACKAGES HAD OCCURRED INSIDE THE DECLARATIVE PART FOR Main.

OVERLOADING AND use CLAUSES -- EXAMPLE 2

```
package P1 is
   A : Integer;                                   -- A#1
   function F (X : Float) return Float;           -- F#1
   function F (X : Integer) return Float;         -- F#2
   function G return Integer;                     -- G#1
end P1;

package P2 is
   function A return Float;                       -- A#2
   function F (X : Integer) return Float;         -- F#3
   function F (X : Integer) return Integer;       -- F#4
   G : Integer;                                   -- G#2
end P2;

with P1, P2; use P1, P2;

procedure Main is
begin
   -- A#1 MUST BE REFERRED TO AS P1.A AND P2.A, SINCE THEY CANNOT COEXIST.
   -- F#1 AND F#4 ARE OVERLOADED AND MAY BE REFERRED TO AS F.
   -- F#2 AND F#3 MUST BE REFERRED TO AS P1.F AND P2.F, SINCE THEY CANNOT COEXIST.
   -- G#1 AND G#2 MUST BE REFERRED TO AS P1.G AND P2.G, SINCE THEY CANNOT COEXIST.
end Main;
```

INSTRUCTOR NOTES

TYPE_1_IO AND TYPE_2_IO ARE PACKAGES PROVIDING SEQUENTIAL I/O OPERATIONS FOR TWO DIFFERENT TYPES. (EACH COULD BE AN INSTANCE OF THE PREDEFINED GENERIC PACKAGE Sequential_IO.) ONLY A FEW REPRESENTATIVE DECLARATIONS ARE SHOWN FOR EACH PACKAGE. THESE DECLARATIONS ARE THE SAME IN EACH PACKAGE.

THE X IN Type_1_IO.X REFERS TO ANY ENTITY DECLARED IN PACKAGE Type_1_IO.

ANSWERS:

IDENTICAL TYPE NAMES CANNOT COEXIST, SO File_Type MUST BE NAMED AS Type_1_IO.File_Type OR Type_2_IO.File_Type AND File_Mode MUST BE NAMED AS Type_1_IO.File_Mode OR Type_2_IO.File_Mode.

BECAUSE THE ENUMERATION LITERAL In_File IN Type_1_IO IS OF TYPE Type_1_IO.File_Mode AND THE CORRESPONDING LITERAL IN Type_2_IO IS OF TYPE Type_2_IO.File_Mode, THE TWO LITERALS HAVE DIFFERENT PROFILES. (NOTE THE HINT.) THEREFORE, THEY ARE OVERLOADED, THAT IS, Main MAY REFER TO THE SIMPLE NAME In_File AND THE MEANING WILL BE DETERMINED FROM THE CONTEXT. SIMILAR REASONING APPLIES TO THE LITERAL Out_File.

THE TWO VERSIONS OF Open DIFFER IN THE TYPES OF THEIR FIRST AND SECOND PARAMETERS, SO THEY ARE ALSO OVERLOADED. (AGAIN, NOTE THE HINT.) Open CAN BE CALLED USING ITS SIMPLE NAME.

IDENTICAL EXCEPTION NAMES CANNOT COEXIST, SO Device_Error MUST BE NAMED EITHER AS Type_1_IO.Device_Error OR Type_2_IO.Device_Error. (THIS IS IRONIC SINCE THESE ARE BOTH RENAMINGS OF THE SAME EXCEPTION, IO_Exceptions.Device_Error.)

THE ACID TEST!

```
with IO_Exceptions;

package Type_1_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Out_File);
   ...
   procedure Open
      (File : in out File_Type;
       Mode : in File_Mode;
       Name : in String;
       Form : in String := "");
   ...
   Device_Error :
       exception renames
          IO_Exceptions.Device_Error;
   ...
end Type_1_IO;
```

```
with IO_Exceptions;

package Type_2_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Out_File);
   ...
   procedure Open
      (File : in out File_Type;
       Mode : in File_Mode;
       Name : in String;
       Form : in String := "");
   ...
   Device_Error :
       exception renames
          IO_Exceptions.Device_Error;
   ...
end Type_2_IO;
```

```
with Type_1_IO, Type_2_IO;
use Type_1_IO, Type_2_IO;
procedure Main is
   ...
begin
   -- WHICH OF THE FOLLOWING IDENTIFIERS ARE OVERLOADED?
   -- WHICH MUST BE NAMED WITH EXPANDED NAMES (E.G. Type_1_IO.X)?
   -- (HINT:  Type_1_IO.File_Mode AND Type_2_IO.File_Mode ARE TWO DISTINCT
   -- TYPES.)

   -- File_Type
   -- File_Mode
   -- In_File
   -- Out_File
   -- Open
   -- Device_Error

end Main;
```

INSTRUCTOR NOTES

IMPLICIT IN POINT 5 IS THAT WE CONSIDER WHETHER A CALL IS A FUNCTION CALL OR A SUBPROGRAM CALL.

THE FOLLOWING FIVE SLIDES SHOW EXAMPLES OF SUCCESSFUL OVERLOAD RESOLUTION OF SUBPROGRAM CALLS.

OVERLOAD RESOLUTION

USUALLY, WHEN AN OVERLOADED SUBPROGRAM NAME IS USED IN A SUBPROGRAM CALL, THE CONTEXT AND THE ACTUAL PARAMETERS OF THE CALL WILL BE SUFFICIENT TO UNIQUELY DETERMINE WHICH OVERLOADED SUBPROGRAM IS TO BE INVOKED.

THE FOLLOWING CRITERIA ARE USED IN SELECTING WHICH SUBPROGRAM TO INVOKE:

1. THE SUBPROGRAM NAME.
2. THE NUMBER OF ACTUAL PARAMETERS IN THE CALL.
3. THE BASE TYPES AND ORDER OF THE ACTUAL PARAMETERS.
4. THE NAMES OF THE FORMAL PARAMETERS (IF NAMED ASSOCIATIONS ARE USED).
5. THE RESULT BASE TYPE (FOR FUNCTIONS).

IF THESE CRITERIA ARE NOT SUFFICIENT TO DETERMINE EXACTLY ONE SUBPROGRAM TO INVOKE, THEN THE SUBPROGRAM CALL IS <u>AMBIGUOUS</u>, AND THUS ILLEGAL.

INSTRUCTOR NOTES

S#1 IS NAMED P1.S AND S#2 IS NAMED P2.S. UNLESS EXPANDED NAMES ARE USED, THERE IS NO WAY TO DISTINGUISH WHICH VERSION IS CALLED.

OVERLOAD RESOLUTION BY SUBPROGRAM NAME

- EXAMPLE OF OVERLOAD RESOLUTION DUE TO THE SUBPROGRAM NAME:

```
declare
   package P1 is                                                       -- S#1
      procedure S (A : in Integer; B : in Integer := 0);
   end P1;

     PROFILE : | PARAMETER BASE TYPES : (1) Integer; (2) Integer |
               | RESULT TYPE          : NONE                     |

   package P2 is                                                       -- S#2
      procedure S (A : in Integer);
   end P2;

     PROFILE : | PARAMETER BASE TYPES : (1) Integer |
               | RESULT TYPE          : NONE        |

   package body P1 is
      ...
   end P1;

   package body P2 is
      ...
   end P2;

   use P1, P2; -- OVERLOADS THE TWO VERSIONS OF S, SINCE THEIR PROFILES DIFFER

begin
   S(1);        -- AMBIGUOUS, MAY INVOKE S#1 WITH B DEFAULTED OR S#2.
   P1.S(1);     -- INVOKES S#1 WITH SECOND PARAMETER DEFAULTED
   P2.S(1);     -- INVOKES S#2
end;
```

INSTRUCTOR NOTES

POINT OUT THE DEFAULT VALUE FOR Y IN P#2. THERE IS NO WAY TO CALL P#1 UNAMBIGUOUSLY.

OVERLOAD RESOLUTION BY NUMBER OF PARAMETERS

```
declare

  procedure P (X : in out Natural);                              -- P#1.
  -- PROFILE:
  --PARAMETER BASE TYPES : (1) Integer
  --RESULT TYPE          : NONE

  procedure P (X : in out Natural; Y : in Natural := 0);         -- P#2.
  -- PROFILE:
  --PARAMETER BASE TYPES : (1) Integer; (2) Integer
  --RESULT TYPE          : NONE

  I, J : Integer := 1;
  ...

begin

  P (I);      -- AMBIGUOUS:  MAY INVOKE P#2 WITH Y DEFAULTED, OR P#1.
  P (I, J);   -- INVOKES P#2.

end;
```

INSTRUCTOR NOTES

WALK THROUGH THE EXAMPLES.

ONLY THE BASE TYPES ARE CONSIDERED, NOT THE SUBTYPES.

OVERLOAD RESOLUTION BY PARAMETER TYPES

```
declare

     procedure P (X : in out Natural; Y : in out Natural);  -- P#1.

       -- PROFILE:
       --|PARAMETER BASE TYPES : (1) Integer; (2) Integer|
       --|RESULT TYPE          : NONE                     |

     procedure P (X : in out Natural; Y : in out Float);    -- P#2.

       -- PROFILE:
       --|PARAMETER BASE TYPES : (1) Integer; (2) Float  |
       --|RESULT TYPE          : NONE                    |

     procedure P (X : in out Float; Y : in out Natural);    -- P#3.

       -- PROFILE:
       --|PARAMETER BASE TYPES : (1) Float; (2) Integer  |
       --|RESULT TYPE          : NONE                    |

     I : Positive := 1;        -- BASE TYPE     :  Integer.
     J : Natural := 1;         -- BASE TYPE     :  Integer.
     F : Float := 0.0;         -- BASE TYPE     :  Float.
     ...

begin

     P (I, J);                 -- INVOKES P#1.
     P (I, F);                 -- INVOKES P#2.
     P (F, I);                 -- INVOKES P#3.

end;
```

INSTRUCTOR NOTES

A POSSIBLE SOURCE OF CONFUSION IS THE DISTINCTION BETWEEN A PARENT TYPE AND A BASE TYPE. MAKE SURE THE CLASS REALIZES THAT THE DERIVED TYPE DECLARATION INTRODUCES A NEW BASE TYPE.

OVERLOAD RESOLUTION BY FORMAL PARAMETER NAMES

```
declare

     type Offset_Type is new Integer;

     procedure P (X1 : in Offset_Type; Y : in out Float);       -- P#1.

            -- PROFILE:
            -- PARAMETER BASE TYPES : (1) Offset_Type; (2) Float
            -- RESULT TYPE          : NONE

     procedure P (X2 : in Natural; Y : in out Float);      -- P#2.

            -- PROFILE:
            -- PARAMETER BASE TYPES : (1) Integer; (2) Float
            -- RESULT TYPE          : NONE

     F  : Float := 0.0;
     ...

begin

     P (1, F);                 -- ILLEGAL; AMBIGUOUS SINCE 1 IS A LITERAL FOR
                               --    BOTH TYPES Offset_Type AND Integer.
     P (X1 =>1, Y =>F);        -- INVOKES P#1.
     P (X2 =>1, Y =>F);        -- INVOKES P#2.

end;
```

INSTRUCTOR NOTES

THIS FOIL IS SELF EXPLANATORY.

OVERLOAD RESOLUTION BY RESULT TYPE

```
declare

     function F (X : Natural) return Positive;                   -- F#1.

          -- PROFILE:
          -- PARAMETER BASE TYPES : (1) Integer
          -- RESULT TYPE          : Integer

     function F (X : Natural) return Character;                  -- F#2.

          -- PROFILE:
          -- PARAMETER BASE TYPES : (1) Integer
          -- RESULT TYPE          : Character

     I : Positive := 1;
     C : Character := 'A';
     ...

begin

     I := F (I);        -- INVOKES F#1.
     C := F (I);        -- INVOKES F#2.
     C := F (F(I));     -- INVOKES F#1 WITH I, THEN F#2 WITH F(I).

end;
```

INSTRUCTOR NOTES

RECAP

- WHEN CAN TWO SUBPROGRAMS OR ENUMERATION LITERALS BE OVERLOADED?
  - ONLY WHEN THEY HAVE DIFFERENT PROFILES
  - TWO SUBPROGRAMS WITH DIFFERENT FORMAL PARAMETER NAMES OR MODES, BUT THE SAME PROFILE, MAY NOT BE OVERLOADED

- WHAT ASPECTS OF A CALL CAN BE USED TO RESOLVE OVERLOADING?
  - SUBPROGRAM NAME
  - NUMBER AND BASE TYPES OF ACTUAL PARAMETERS
  - FORMAL PARAMETER NAMES IN NAMED CALLS
  - RESULT BASE TYPE, IF ANY

- THE FOLLOWING ARE NEVER RELEVANT IN ALLOWING OR RESOLVING OVERLOADING:
  - PARAMETER MODES (in, out, in out)
  - PARAMETER SUBTYPES

- SOMETIMES AMBIGUITY CAN'T BE AVOIDED EVEN WHEN PROFILES ARE DIFFERENT:

```
procedure P (X : in Integer; Y : in Integer := 0);       -- P#1

   -- PROFILE:
   --|PARAMETER BASE TYPES : (1) Integer; (2) Integer|
   --|RESULT TYPE          : NONE                     |

procedure P (X : in Integer);                            -- P#2

   -- PROFILE:
   --|PARAMETER BASE TYPES : (1) Integer              |
   --|RESULT TYPE          : NONE                     |

  -- P#2 CANNOT BE CALLED UNAMBIGUOUSLY
  -- THOUGH OVERLOADING IS ALLOWED, IT IS USELESS
```

INSTRUCTOR NOTES

THE FOLLOWING SIX SLIDES SHOW EXAMPLES OF THESE TECHNIQUES FOR CORRECTING AMBIGUOUS SUBPROGRAM CALLS.

PROGRAMMING TIPS -- ELIMINATING AMBIGUOUS CALLS

WHEN OVERLOAD RESOLUTION OF A SUBPROGRAM CALL RESULTS IN AN AMBIGUOUS CALL, YOU CAN ELIMINATE THE AMBIGUITY BY MAKING ONE OR MORE OF THE FOLLOWING CHANGES TO THE CALL:

1. ENCLOSE ONE OR MORE ACTUAL PARAMETERS IN A QUALIFIED EXPRESSION.

2. ENCLOSE A FUNCTION CALL IN A QUALIFIED EXPRESSION.

3. USE THE FORMAL PARAMETER NAMES IN NAMED ASSOCIATIONS WITH THE ACTUAL PARAMETERS.

4. USE AN EXPANDED NAME FOR THE SUBPROGRAM (E.G., Package.Subprogram).

5. RENAME THE SUBPROGRAM VIA A RENAMING DECLARATION.

6. EXPLICITLY PROVIDE PARAMETER VALUES INSTEAD OF USING DEFAULTS.

INSTRUCTOR NOTES

THIS FOIL IS SELF EXPLANATORY.

QUALIFIED EXPRESSIONS TO AVOID AMBIGUITY

RESOLVING AN AMBIGUOUS SUBPROGRAM CALL BY QUALIFYING THE TYPE OF ONE OR MORE ACTUAL PARAMETERS:

```
declare
     type Fruit_Type is (Apple, Orange, Lemon, Lime, Grape);
     type Flavor_Type is (Grape, Orange, Lemon_Lime);
     procedure P (X : in Fruit_Type; Y : out Natural);              -- P#1.
          -- PROFILE:
          --|PARAMETER BASE TYPES : (1) Fruit_Type; (2) Integer
          --|RESULT TYPE          : NONE
     procedure P (X : in Flavor_Type; Y : out Natural);             -- P#2.
          -- PROFILE:
          --|PARAMETER BASE TYPES : (1) Flavor_Type; (2) Integer
          --|RESULT TYPE          : NONE
     I : Natural;  -- BASE TYPE:  Integer.
     ...
begin
     P (Orange, I);                       -- ILLEGAL; AMBIGUOUS SINCE Orange IS A
                                          --    LITERAL FOR BOTH TYPES Fruit_Type AND
                                          --    Flavor_Type.
     P (Fruit_Type'(Orange), I);          -- INVOKES P#1.
     P (Flavor_Type'(Orange), I);         -- INVOKES P#2.
end;
```

INSTRUCTOR NOTES

THE OPERATOR < IS DEFINED FOR ENUMERATION TYPES.

QUALIFIED EXPRESSIONS TO AVOID AMBIGUITY

```
declare

     type Fruit_Type is (Apple, Orange, Lemon, Lime, Grape);
     type Flavor_Type is (Grape, Orange, Lemon_Lime);

     function F (X : Natural) return Fruit_Type; -- F#1.

           -- PROFILE:
           -- PARAMETER BASE TYPES : (1) Integer
           -- RESULT TYPE          : Fruit_Type

     function F (X : Natural) return Flavor_Type;     -- F#2.

           -- PROFILE:
           -- PARAMETER BASE TYPES : (1) Integer
           -- RESULT TYPE          : Flavor_Type

     B : Boolean;
     ...

begin

     B := F (1) < F (2);                   -- ILLEGAL; AMBIGUOUS SINCE BOTH F'S
                                           --    CAN RETURN EITHER TYPE Fruit_Type OR
                                           --    Flavor_Type AND < IS DEFINED FOR
                                           --    BOTH TYPES.
     B := Fruit_Type'(F(1)) < F (2);       -- BOTH F'S ARE F#1.
     B := F (1) < Flavor_Type'(F(2));      -- BOTH F'S ARE F#2.

end;
```

INSTRUCTOR NOTES

THIS FOIL IS SELF EXPLANATORY.

NAMED SUBPROGRAM CALLS TO AVOID AMBIGUITY

```
declare

     procedure P (X : in out Natural);                         -- P#1.

            -- PROFILE:
            -- PARAMETER BASE TYPES : (1) Integer
            -- RESULT TYPE          : NONE

     procedure P (Y : in out Natural; X : in Natural := 0);     -- P#2.

            -- PROFILE:
            -- PARAMETER BASE TYPES : (1) Integer; (2) Integer
            -- RESULT TYPE          : NONE

     I : Natural := 1;        -- BASE TYPE:  Integer.
     ...

begin

     P (I);          -- ILLEGAL; AMBIGUOUS SINCE P#2 HAS A DEFAULT VALUE FOR
                     --    ITS SECOND PARAMETER.
     P (Y => I);     -- INVOKES P#2.
     P (X => I);     -- INVOKES P#1 SINCE PARAMETER Y OF P#2 HAS NO DEFAULT
                     --    VALUE.

end;
```

INSTRUCTOR NOTES

THIS FOIL IS SELF EXPLANATORY.

EXPANDED NAMES TO AVOID AMBIGUITY

```
declare

     package Package_1 is
          procedure P (X : in out Natural);                         -- P#1.

            -- PROFILE:
            -- PARAMETER BASE TYPES : (1) Integer
            -- RESULT TYPE          : NONE

     end Package_1;

     package Package_2 is
          procedure P (X : out Natural; Y : in Natural := 0);   -- P#2.

            -- PROFILE:
            -- PARAMETER BASE TYPES : (1) Integer; (2) Integer
            -- RESULT TYPE          : NONE

     end Package_2;

     use Package_1, Package_2;    -- MAKES BOTH P#1 AND P#2 DIRECTLY VISIBLE
                                  --    SINCE THEY HAVE DIFFERENT PROFILES.
     I : Natural := 1;            -- BASE TYPE:  Integer.
     ...

begin

     P (I);                    -- ILLEGAL; AMBIGUOUS SINCE P#2'S SECOND PARAMETER HAS
                               --    A DEFAULT VALUE.
     Package_1.P (I);          -- INVOKES P#1.
     Package_2.P (I);          -- INVOKES P#2.

end;
```

INSTRUCTOR NOTES

THIS FOIL IS SELF EXPLANATORY.

RENAMING TO AVOID AMBIGUITY

EXAMPLE OF RESOLVING AN AMBIGUOUS SUBPROGRAM CALL BY RENAMING THE SUBPROGRAM VIA A RENAMING DECLARATION:

```
procedure P (X : in out Natural);                           -- P#1.

          -- PROFILE:
          -- PARAMETER BASE TYPES : (1) Integer
          -- RESULT TYPE          : NONE

...

declare

     procedure P1 (X : in out Natural) renames P;           -- P#1.

     procedure P (X : out Natural; Y : in Natural := 0);    -- P#2.

          -- PROFILE:
          -- PARAMETER BASE TYPES : (1) Integer; (2) Integer
          -- RESULT TYPE          : NONE

     I : Natural := 1;  -- BASE TYPE:  Integer.
     ...

begin

     P (I);   -- ILLEGAL; AMBIGUOUS SINCE P#2'S SECOND PARAMETER HAS
              --    A DEFAULT VALUE.
     P1 (I);  -- INVOKES P#1.

end;
```

INSTRUCTOR NOTES

THE SECOND PARAMETER IS GIVEN EXPLICITLY TO RESOLVE OVERLOADING EVEN THOUGH THE DEFAULTS GIVE THE DESIRED VALUES.

EXPLICIT PARAMETERS TO AVOID AMBIGUITY

EXAMPLE OF RESOLVING AN AMBIGUOUS SUBPROGRAM CALL BY EXPLICITLY PROVIDING DEFAULTABLE PARAMETERS:

```
declare
   procedure P (X : out Natural; Y : in Integer := 0);       -- P#1

            -- PROFILE:
            --|PARAMETER BASE TYPES : (1) Integer; (2) Integer|
            --|RESULT TYPE          : NONE                    |

   procedure P (X : out Natural; Y : in Float := 0.0);       -- P#2

            -- PROFILE:
            --|PARAMETER BASE TYPES : (1) Integer; (2) Float  |
            --|RESULT TYPE          : NONE                    |

   N : Natural;
   ...
begin

   P(N);      -- ILLEGAL; AMBIGUOUS SINCE P#1 AND P#2 BOTH HAVE DEFAULT VALUES FOR Y.
   P(N, 0);   -- INVOKES P#1. (SECOND ACTUAL PARAMETER IS AN INTEGER LITERAL.)
   P(N, 0.0); -- INVOKES P#2. (SECOND ACTUAL PARAMETER IS A REAL LITERAL.)

end;
```

INSTRUCTOR NOTES

OVERVIEW.

OVERLOADING OF OPERATORS

- IN Ada, OPERATORS SUCH AS + AND <= ARE ACTUALLY FUNCTIONS.
  - LEFT AND RIGHT OPERANDS ARE ACTUAL PARAMETERS
  - VALUE RETURNED BECOMES VALUE OF THE EXPRESSION CONTAINING THE OPERATOR

- OPERATORS OBEY ALL THE RULES AND HAVE ALL THE PROPERTIES OF ORDINARY FUNCTIONS, INCLUDING BEING USER-DEFINABLE AND OVERLOADABLE.

- MAIN DIFFERENCES:

1. THE NAME, OR DESIGNATOR, OF AN OPERATOR'S FUNCTION LOOKS LIKE A STRING LITERAL RATHER THAN AN IDENTIFIER (E.G., "+" AND "<=").

2. OPERATORS MAY BE INVOKED EITHER WITH FUNCTION CALL SYNTAX, E.G.,

```
"+" (X, Y)              "+" (X, "*"(Y, Z))
```

   OR, EQUIVALENTLY, WITH OPERATOR/OPERAND SYNTAX, E.G.,

```
X + Y                   X + Y * Z
```

INSTRUCTOR NOTES

THIS EXERCISE IS MEANT TO UNCOVER ANY MISUNDERSTANDINGS ABOUT HOW TO VIEW OPERATORS AS FUNCTIONS AND TO ACT AS FOCUS FOR QUESTIONS. THE EXERCISE SHOULD SENSITIZE STUDENTS TO THE IMPORTANCE OF PRECEDENCE RULES AND LEFT-TO-RIGHT APPLICATION OF OPERATORS WITHIN A PRECEDENCE LEVEL.

EMPHASIZE THAT, WHILE BOTH FORMS ARE LEGAL, THE OPERATOR FORM IS STYLISTICALLY PREFERABLE.

```
ANSWERS:     A + B + C
             A - (B - C)              (PARENTHESES ARE CRUCIAL!)
             (A +B) * C               (PARENTHESES ARE CRUCIAL!)
             A < B and B < C
             (A and B) or (C and D)   (PARENTHESES REQUIRED BY SYNTAX)

             "*" ("*" (X, Y), Z)
             "=" ("+" (A, B), "+" (C, D))
             "*" (I, "**" *J, K))
             "or" ("=" (A, 0), ">" *"+" (B, C), D))
             "=" ("mod" *Y, 12), 0)
```

OPERATORS AS FUNCTIONS

- REWRITE THE FOLLOWING IN TRADITIONAL OPERATOR NOTATION:

"+" ("+" (A, B), C) ____________________

"-" (A, "-" (B, C)) ____________________

"*" ("+" (A, B), C) ____________________

"and" ("<" (A, B), "<" (B, C)) ____________________

"or" ("and" (A, B), "and" (C, D)) ____________________

- REWRITE THE FOLLOWING IN FUNCTION CALL NOTATION:

X*Y*Z ____________________

A+B = C+D ____________________

I*J**K ____________________

A=0 or B+C>D ____________________

Y mod 12 = 0 ____________________

INSTRUCTOR NOTES

OPERATORS AND FUNCTIONS

- THE FOLLOWING OPERATORS HAVE CORRESPONDING FUNCTIONS:
  - FUNCTIONS WITH TWO PARAMETERS

    ```
    and  or  xor  =  /=  <  >  <=  >=  &  *  /  mod  rem  **
    ```

  - FUNCTIONS WITH ONE PARAMETERS:

    ```
    not abs
    ```

  - FUNCTIONS WITH OVERLOADED ONE- AND TWO-PARAMETER VERSIONS:

    ```
    + -
    ```

- THESE FUNCTIONS ARE NAMED, OR DESIGNATED, BY ENCLOSING THE OPERATOR IN QUOTATION MARKS:

  ```
  "and"  "<="  "+"  "abs"
  ```

- THERE ARE NO FUNCTIONS CORRESPONDING TO THE FOLLOWING:

  ```
  and then, or else, in, not in
  ```

- THE CASE (UPPER OR LOWER) OF LETTERS IS IGNORED FOR OPERATORS, SO THAT, FOR EXAMPLE, THE OPERATOR ABS CAN BE WRITTEN AS

  ```
  abs   Abs   ABS   aBs
  ```

  AND ITS CORRESPONDING FUNCTION DESIGNATOR CAN BE WRITTEN AS

  ```
  "abs"  "Abs"  "ABS"  "aBs"
  ```

  AND ALL OF THESE DENOTE THE SAME OPERATOR OR FUNCTION.

INSTRUCTOR NOTES

SEE APPENDIX C OF THE LRM.

DECLARATION OF PREDEFINED TYPES

THE PACKAGE Standard CONTAINS THE DECLARATIONS OF MOST OF THE PREDEFINED TYPES AND OPERATORS, E.G.,

```
type Boolean is (False, True);

function "<="  (Left, Right : Boolean) return Boolean;
function "and" (Left, Right : Boolean) return Boolean;

type Integer is ...;

function "+"   (      Right : Integer) return Integer;     -- UNARY PLUS, E.G. +65
function "+"   (Left, Right : Integer) return Integer;     -- BINARY PLUS, E.G. 3+4
function "mod" (Left, Right : Integer) return Integer;
function ">"   (Left, Right : Integer) return Boolean;
```

NOTE THAT THESE PREDEFINED OPERATOR FUNCTIONS ARE OVERLOADED FOR ALL THE APPROPRIATE PREDEFINED TYPES, E.G., "+" FOR ALL NUMERIC TYPES, "=" FOR ALL NON-LIMITED TYPES.

INSTRUCTOR NOTES

THE OTHER RELATIONAL OPERATORS (<, >, <=, AND >=) ARE NOT IMPLICITLY DECLARED FOR TYPE VECTOR BECAUSE THE COMPONENT TYPE OF Vector_Type IS Float, WHICH IS NOT A DISCRETE TYPE.

IMPLICIT DECLARATIONS

A TYPE DECLARATION IMPLICITLY DECLARES THE OPERATOR FUNCTIONS THAT ARE APPROPRIATE FOR THAT CLASS OF TYPE (I.E., DISCRETE, NUMERIC, ARRAY, RECORD, ACCESS, PRIVATE, ETC.). FOR EXAMPLE,

```
type Vector_Type is array (Positive range <>) of Float;
```

IMPLICITLY DECLARES THE FOLLOWING OPERATORS:

```
function "="  (Left, Right : Vector_Type) return Boolean;
function "/=" (Left, Right : Vector_Type) return Boolean;

function "&" (Left : Vector_Type; Right : Vector_Type) return Vector_Type;
function "&" (Left : Vector_Type; Right : Float      ) return Vector_Type;
function "&" (Left : Float ;      Right : Vector_Type) return Vector_Type;
function "&" (Left : Float ;      Right : Float      ) return Vector_Type;
```

THESE IMPLICIT DECLARATIONS PROVIDE ADDITIONAL OVERLOADINGS OF THE OPERATORS =, /=, AND &.

WALK THROUGH THE EXAMPLES.

RESOLUTION OF OVERLOADED OPERATORS

RESOLUTION OF OVERLOADED OPERATORS IN EXPRESSIONS FOLLOWS THE SAME RULES AS FOR ORDINARY FUNCTIONS.

```
declare

     type Vector is array (Positive range <>) of Float;
     -- IMPLICIT DECLARATIONS FOR =, /=, AND & FOR TYPE Vector.
     F1, F2 : Float;
     V1, V2 : Vector (1 .. 3);
     V3     : Vector (1 .. 6);
     S1, S2 : String (1 .. 3);
     S3     : String (1 .. 6);
     B      : Boolean;

begin

     ...
     B := F1 = F2;                     -- RESOLVES TO "=" FOR TYPE Float.
     B := V1 = V2;                     -- RESOLVES TO "=" FOR TYPE Vector.
     B := "=" (V1, V2);                -- RESOLVES TO "=" FOR TYPE Vector.
     V3 := V1 & V2;                    -- RESOLVES TO "&" FOR TYPE Vector.
     S3 := S1 & S2;                    -- RESOLVES TO "&" FOR TYPE String.
     S3 := Standard."&" (S1, S2);      -- RESOLVES TO "&" FOR TYPE String.

     S3 := S1 Standard."&" S2;         -- ILLEGAL EXPRESSION (OPERATOR) SYNTAX!

end;
```

INSTRUCTOR NOTES

EMPHASIZE THE BENEFITS OF WRITING V1 + V2 INSTEAD OF SOMETHING LIKE Add_Vector (V1, V2);.

DECLARATION OF OPERATORS

- YOU CAN DECLARE ADDITIONAL OPERATORS FOR A TYPE BY SUPPLYING APPROPRIATE FUNCTION DECLARATIONS AND BODIES.

- THIS OVERLOADS OTHER MEANINGS OF THE OPERATOR.

- EXAMPLE:

```
package Vector_Package is
   type Vector_Type is private;
   ...
   function "+" (Left, Right : Vector_Type) return Vector_Type;
   ...
private
   type Vector_Type is array (1 .. 3) of Float;
end Vector_Package;

package body Vector_Package is
   ...
   function "+" (Left, Right : Vector_Type) return Vector_Type is
      Result : Vector_Type;
   begin
      for I in 1 .. 3 loop
         Result (I) := Left (I) + Right (I);
                               -- RESOLVES TO "+" FOR TYPE Float.
      end loop;
   end "+";
end Vector_Package;

with Vector_Package; use Vector_Package;
procedure Main is
   V1, V2 : Vector_Type;
   ...
begin
   ...
   V1 := V1 + V2; -- RESOLVES TO "+" FOR Vector_Type
   ...
end Main;
```

INSTRUCTOR NOTES

HIDING AN OPERATOR DECLARATION

YOU CAN HIDE OR REPLACE AN EXISTING OPERATOR DECLARATION BY EXPLICITLY GIVING ANOTHER OPERATOR DECLARATION THAT HAS THE SAME PARAMETER AND RESULT TYPE PROFILE.

```
B : declare

        subtype Hour_Subtype is Integer range 0 .. 23;
        Hour : Hour_Subtype := 0;

        function "+" (Left, Right : Hour_Subtype) return Hour_Subtype is      -- B."+".

              -- PROFILE:
              -- PARAMETER BASE TYPES : (1) Integer; (2) Integer
              -- RESULT TYPE          : Integer

        begin -- "+"
                    return Standard."+" (Left, Right) mod 24;
                    -- Note that writing:  return (Left + Right) mod 24;
                    --     would invoke B."+" recursively.
        end "+";

    begin -- Block B

        Hour := Hour + 1;  -- INVOKES B."+".

    end B;
```

INSTRUCTOR NOTES

OVERLOADING RULES FOR OPERATORS

EXCEPT FOR = AND /=, THERE ARE NO SPECIAL RULES THAT RESTRICT THE TYPES OF THE OPERANDS AND RESULTS OF OPERATORS.

FOR EXAMPLE, THE OPERATOR & IS IMPLICITLY DECLARED FOR ALL COMBINATIONS OF A ONE-DIMENSIONAL ARRAY TYPE AND ITS COMPONENT TYPE:

```
function "&" (Left : String;      Right : String)     return String
function "&" (Left : Character;   Right : String)     return String
function "&" (Left : String;      Right : Character)  return String
function "&" (Left : Character;   Right : Character)  return String
```

INSTRUCTOR NOTES

SOME RESTRICTIONS ON OPERATOR DECLARATIONS

1. YOU CANNOT CREATE NEW OPERATOR SYMBOLS, E.G., \ OR SORT.

2. YOU CANNOT CHANGE THE PRECEDENCE OF OPERATORS.

3. THE UNARY OPERATORS not AND abs CAN HAVE ONLY ONE PARAMETER.

4. THE BINARY OPERATORS and, or, xor, =, < , > , <=, >=, &, *, /, mod, rem, AND ** CAN HAVE ONLY TWO PARAMETERS.

5. THE OPERATORS + AND - MUST HAVE EITHER ONE OR TWO PARAMETERS.

6. THE OPERATOR /= CANNOT BE EXPLICITLY DECLARED. IT IS IMPLICITLY DECLARED BY A DECLARATION OF = TO BE THE OPPOSITE OF =.

7. THE FOLLOWING BASIC OPERATIONS ARE NOT USER-DEFINABLE: and then, or else, in, not in, :=, SELECTION, INDEXING, SLICING, QUALIFICATION, CONVERSION, ATTRIBUTION, AND ALLOCATION.

8. DEFAULT EXPRESSIONS ARE NOT ALLOWED FOR THE PARAMETERS OF OPERATORS.

INSTRUCTOR NOTES

THE VARYING-LENGTH STRING PACKAGE WAS COVERED IN III.B.4.

ADDITIONAL RESTRICTIONS ON DECLARATIONS OF THE = OPERATOR

1. BOTH PARAMETERS MUST BE OF THE SAME LIMITED TYPE.

```
function "=" (Left : Integer; Right : Float) return Boolean;     -- **ILLEGAL
```

2. THE RESULT TYPE MUST BE THE PREDEFINED TYPE BOOLEAN.

```
function "=" (Left, Right : Varying_String_Type) return Integer; -- **ILLEGAL
```

3. A RENAMING DECLARATION THAT DECLARES "=" MUST RENAME "=" TO ANOTHER EQUALITY OPERATOR.

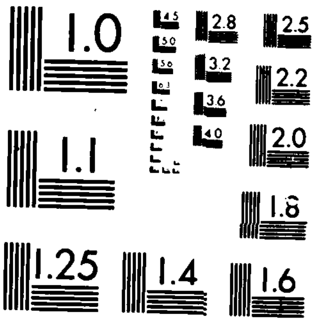
```
function "=" (Left, Right : Varying_String_Type) return Boolean renames
        Varying_Strings_Package.Less_Than;                        -- **ILLEGAL
```

INSTRUCTOR NOTES

A REVISIT OF Package_Varying_Strings_Package WHERE SOME OPERATIONS ARE NOW OVERLOADED OPERATORS.

AD-A165 076 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506 2/7

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

PACKAGE Varying_Strings_Package - REVISITED

```
package Varying_Strings_Package is
   ...
   function Length (...) return ...;
   function Content (...) return ...;
   procedure Copy (Source : Varying_String_Type; ...);
   procedure Copy (Source : String;   ...);                -- REPLACES INITIALIZE.
   procedure Append (Source : Varying_String_Type; ...);
   procedure Append (Source : String;   ...);                   -- NEW.
   procedure Change (...; Source : Varying_String_Type);
   procedure Change (...; Source : String);                     -- NEW.
   procedure Substring (...; Target : in out Varying_String_Type);
   function  Substring (...) return String;                     -- NEW.
   function "=" (Left, Right : Varying_String_Type) return ...; -- REPLACES Equal.
   function "<" (Left, Right : Varying_String_Type) return ...; -- REPLACES Less_Than.
   function ">" (Left, Right : Varying_String_Type) return ...; -- REPLACES Greater_Than.
   function Equal (...; Right : String) return ...;             -- NEW.
   function "<"   (...; Right : String) return ...;             -- NEW.
   function ">"   (...; Right : String) return ...;             -- NEW.
   function Next (Pattern : Varying_String_Type; ...) return ...;
   function Next (Pattern : String;   ...) return ...;          -- NEW.
   ...
private
   ...
end Varying_Strings_Package;
```

ANOTHER EXAMPLE.

RATIONAL NUMBERS

A LARGER EXAMPLE OF AN ABSTRACT NUMERIC DATA TYPE WHOSE SPECIFICATION AND IMPLEMENTATION OVERLOADS THE NUMERIC AND RELATIONAL OPERATORS.

INFORMAL DESCRIPTION OF RATIONAL NUMBERS:

- IN MATHEMATICS, A RATIONAL NUMBER IS A REAL NUMBER THAT CAN BE EXACTLY REPRESENTED AS THE QUOTIENT OF TWO INTEGERS, E.G., 5/3.
- THE USUAL NUMERIC OPERATIONS +, -, *, AND /, IF GIVEN RATIONAL OPERANDS, WILL YIELD RATIONAL RESULTS.
- THE USUAL RELATIONAL OPERATIONS =, /=, < , >, <=, AND >= ARE ALSO APPLICABLE TO RATIONAL NUMBERS.
- RATIONAL NUMBERS ARE USEFUL WHEN AN EXACT REPRESENTATION (NO ROUNDING OR TRUNCATION) OF NON-INTEGRAL NUMBERS IS NEEDED. FOR EXAMPLE, 5/3 CAN ONLY BE APPROXIMATELY REPRESENTED BY Ada'S FIXED AND FLOATING POINT TYPES.

INSTRUCTOR NOTES

THIS IS THE OVERALL STRUCTURE OF THE SPECIFICATION FOR THE RATIONAL ABSTRACT DATA TYPE. THE ... PORTIONS ARE EXPANDED IN THE FOLLOWING SLIDES.

ABSTRACTION FOR TYPE RATIONAL

```
-- SPECIFICATION OF ABSTRACTION FOR TYPE RATIONAL.

    package Rational_Numbers is

         -- TYPES, SUBTYPES, AND CONSTANTS.

              type Rational_Type is ...;
              ...

         -- OPERATIONS.

              function Make_Rational (Numerator : Integer; Denominator : Positive)
                  return Rational_Type;
              ...
              function "+" (Left, Right : Rational_Type) return Rational_Type;
              ...
              function "<" (Left, Right : Rational_Type) return Boolean;
              ...

         -- EXCEPTIONS.

                    ...

    private
        ...
    end Rational_Numbers;
```

INSTRUCTOR NOTES

NOTE THAT BECAUSE Rational IS A PRIVATE TYPE RATHER THAN A LIMITED PRIVATE TYPE, IT HAS THE FOLLOWING BASIC AND PREDEFINED OPERATIONS IN ADDITION TO THE OPERATIONS WE WILL EXPLICITLY DECLARE FOR IT: :=, =, /=.

IF WE LATER DISCOVER THAT OUR REPRESENTATION FOR TYPE Rational DOES NOT YIELD THE DESIRED SEMANTICS FOR := AND =, THEN WE CAN EITHER CHANGE THE REPRESENTATION OR CHANGE Rational TO BE A LIMITED PRIVATE TYPE AND EXPLICITLY PROVIDE ASSIGNMENT AND EQUALITY OPERATIONS.

ALTHOUGH THIS ABSTRACTION WILL PERMIT A USER TO DECLARE HIS OWN CONSTANTS, THE VALUES ZERO AND ONE ARE NEEDED MORE FREQUENTLY THAN ANY OTHERS, AND SO ARE PROVIDED AS CONSTANTS BY THIS PACKAGE.

GO QUICKLY THROUGH THESE 4 SLIDES.

ABSTRACTION FOR TYPE RATIONAL (Continued)

```
-- TYPES, SUBTYPES, AND CONSTANTS.

     type Rational_Type is private;

     Zero : constant Rational_Type;     -- DEFERRED CONSTANTS
     One  : constant Rational_Type;
```

INSTRUCTOR NOTES

THESE CONSTRUCTION AND SELECTION OPERATIONS PROVIDE A MINIMAL WAY TO COMMUNICATE AND RELATE RATIONAL NUMBERS WITH THE REST OF Ada'S NUMERIC TYPES, AND HENCE WITH THE EXTERNAL WORLD VIA Integer_IO AND/OR Float_IO.

FOR EXAMPLE, A USER MIGHT WANT TO OBTAIN AN APPROXIMATE VALUE FOR A RATIONAL NUMBER BY CONVERTING IT TO SOME FIXED OR FLOATING POINT TYPE, AS IN:

```
type Real is ...;  -- A FIXED OR FLOATING POINT TYPE.

function Rational_To_Real (R : Rational) return Real is
begin -- Rational_To_Real
   return Real (Real (Numerator (R)) / Real (Denominator (R)));
end Rational_To_Real;
```

OPERATIONS ON TYPE Rational

```
-- CONSTRUCTION OPERATIONS.


   function Make_Rational (Numerator : Integer; Denominator : Positive)
      return Rational_Type;

    -- VALUE: THE RATIONAL NUMBER THAT CORRESPONDS TO Numerator / Denominator, WHERE
    --    / IS THE MATHEMATICAL DIVISION OPERATOR (NO TRUNCATING OR
    --    ROUNDING).

 -- Selection operations.

   function Numerator   (R : Rational_Type) return Integer;
   function Denominator (R : Rational_Type) return Positive;

    -- INVARIANT: FOR ALL R IN Rational_Type =>
    --            R = Numerator (R) / Denominator (R),
    --        WHERE = AND / ARE THE MATHEMATICAL OPERATORS.
```

INSTRUCTOR NOTES

THE NUMERIC OPERATORS ARE NOW OVERLOADED FOR TYPE RATIONAL.

THE ABSTRACTION PROVIDES THE SAME NUMERIC OPERATORS FOR TYPE RATIONAL AS Ada PROVIDES FOR ITS REAL TYPES (FLOATING AND FIXED POINT TYPES), SINCE RATIONAL NUMBERS ARE REAL NUMBERS AND IN GENERAL ARE NOT INTEGERS, AND SINCE THE INTEGER OPERATORS "mod" AND "rem" HAVE NO CORRESPONDING MEANING FOR RATIONAL NUMBERS.

NUMERIC OPERATIONS FOR Rational_Type

```
-- NUMERIC OPERATIONS.



   function "+" (Left, Right : Rational_Type) return Rational_Type;
   function "-" (Left, Right : Rational_Type) return Rational_Type;
   function "*" (Left, Right : Rational_Type) return Rational_Type;
   function "/" (Left, Right : Rational_Type) return Rational_Type;

   function "**" (Left : Rational_Type; Right : Integer) return Rational_Type;

   function "+"   (Right : Rational_Type) return Rational_Type;
   function "-"   (Right : Rational_Type) return Rational_Type;
   function "abs" (Right : Rational_Type) return Rational_Type;

    -- THESE OPERATORS YIELD THE CORRESPONDING MATHEMATICAL RESULT
    -- (OR RAISE Numeric_Error).
```

INSTRUCTOR NOTES

THE RELATIONAL OPERATORS ARE NOW OVERLOADED FOR TYPE Rational.

NOTE THAT = AND /= ARE PREDEFINED BECAUSE Rational IS A PRIVATE TYPE.

RELATIONAL OPERATIONS FOR Rational_Type

```
-- RELATIONAL OPERATIONS.


          function "<"  (Left, Right : Rational_Type) return Boolean;
          function ">"  (Left, Right : Rational_Type) return Boolean;
          function "<=" (Left, Right : Rational_Type) return Boolean;
          function ">=" (Left, Right : Rational_Type) return Boolean;

            -- THESE OPERATORS YIELD THE CORRESPONDING MATHEMATICAL RESULT
            -- (OR RAISE Numeric_Error).
```

INSTRUCTOR NOTES

THIS IMPLEMENTATION OF TYPE Rational_Type COMES DIRECTLY FROM THE DEFINITION OF A RATIONAL NUMBER AS BEING THE QUOTIENT OF TWO INTEGERS. SINCE ALL THE Ada DIVISION OPERATORS ARE SUBJECT TO ROUNDING/TRUNCATION, WHEREAS OUR ABSTRACTION REQUIRES AN EXACT MATHEMATICAL RESULT, WE REPRESENT A RATIONAL NUMBER IN ITS UNDIVIDED FORM AS A NUMERATOR AND A DENOMINATOR OF A QUOTIENT.

FOR CLARITY, WE USE THE SIGN OF THE NUMERATOR AS THE SIGN OF A RATIONAL NUMBER, AND THUS KEEP THE DENOMINATOR NON-NEGATIVE. A DENOMINATOR VALUE OF ZERO COULD BE USED TO REPRESENT INFINITY, BUT THE ABSTRACTION REQUIRES THAT Numeric_Error BE RAISED FOR INFINITY. HENCE, THE DENOMINATOR IS KEPT POSITIVE.

WE CHOOSE NOT TO PROVIDE DEFAULT INITIAL VALUES FOR RATIONAL OBJECTS BECAUSE:

1. NONE OF THE Ada NUMERIC TYPES PROVIDE DEFAULT INITIAL VALUES.

2. TYPE Rational_Type HAS AN ASSIGNMENT OPERATION, SINCE IT IS A PRIVATE TYPE, AND A CONSTRUCTOR FUNCTION (Make_Rational), SO THAT A USER MAY EASILY INITIALIZE HIS VARIABLES.

3. AN IMPLEMENTATION OF THE ABSTRACTION NEED NOT BE CONCERNED WITH THE APPLICATION OF THE OPERATIONS OF TYPE Rational_Type TO UNINITIALIZED OBJECTS SINCE SUCH PROGRAMS ARE ERRONEOUS. THE IMPLEMENTATION SHOULD PROTECT ITS INTERNAL DATA, BUT IS NOT REQUIRED TO PRODUCE A SENSIBLE RESULT NOR RAISE AN EXCEPTION.

THE PRIVATE PART OF PACKAGE Rational_Numbers

```
package Rational_Numbers is

   type Rational_Type is private;

   ...

private

  type Rational_Type is
  record
       Numerator_Part   : Integer;      -- Numerator.
       Denominator_Part : Positive;     -- Denominator.
  end record;

  Zero : constant Rational_Type := (0, 1);
  One  : constant Rational_Type := (1, 1);

end Rational_Numbers;
```

INSTRUCTOR NOTES

IMPLEMENTATION OF BINARY OPERATORS

LET L AND R DENOTE THE LEFT AND RIGHT OPERANDS.

LET N AND D DENOTE THE NUMERATOR AND DENOMINATOR COMPONENTS.

$$L + R = \frac{L.N}{L.D} + \frac{R.N}{R.D} = \frac{L.N * R.D}{L.D * R.D} + \frac{R.N * L.D}{R.D * L.D} = \frac{L.N * R.D + R.N * L.D}{L.D * R.D}$$

$$L * R = \frac{L.N}{L.D} * \frac{R.N}{R.D} = \frac{L.N * R.N}{L.D * R.D}$$

INSTRUCTOR NOTES

BULLET 3: "REDUCED" MEANS THE NUMERATOR AND DENOMINATOR *HAVE NO COMMON FACTORS OTHER* THAN 1.

A Greatest Common Divisor ALGORITHM IS NEEDED SO THAT THE = OPERATOR FOR PRIVATE TYPE Rational_Type WILL WORK CORRECTLY VIA THE SECOND METHOD.

REMIND STUDENTS THAT THE ONLY TIME YOU CAN OVERLOAD "=" IS FOR A LIMITED PRIVATE TYPE. ("/=" IS AUTOMATICALLY OVERLOADED WHEN YOU OVERLOAD "="; YOU CAN'T OVERLOAD "/=" EXPLICITLY.)

PROBLEMS WITH PREDEFINED EQUALITY

- IN MATHEMATICS, (16 / 60) AND (4 / 15) ARE EQUIVALENT REPRESENTATIONS FOR THE SAME RATIONAL NUMBER.

  HOWEVER, Ada'S PREDEFINED = OPERATOR FOR PRIVATE TYPE Rational_Type DOESN'T KNOW THAT THESE TWO REPRESENTATIONS SHOULD BE CONSIDERED EQUIVALENT. IT IS REALLY A RECORD TYPE = OPERATOR, AND THE TWO RECORD VALUES

  ```
  Rational_Type'(16, 60)  AND  Rational_Type'(4, 15)
  ```

  ARE DIFFERENT.

- TWO WAYS TO OBTAIN A CORRECT = OPERATOR FOR Rational_Type.

  1. CHANGE Rational_Type TO BE A LIMITED PRIVATE TYPE AND PROVIDE AN EXPLICIT OVERLOADING OF = THAT CHECKS WHETHER ITS OPERANDS REPRESENT THE SAME MATHEMATICAL VALUE.

  2. GUARANTEE THAT EACH ABSTRACT RATIONAL NUMBER IS REPRESENTED BY A UNIQUE Rational_Type RECORD.

- WE CAN IMPLEMENT THE SECOND METHOD BY ALWAYS REPRESENTING AN ABSTRACT RATIONAL NUMBER IN REDUCED FORM WITH A POSITIVE DENOMINATOR.

- REDUCING FRACTIONS WILL ALSO PREVENT Numerator_Part AND Denominator_Part COMPONENTS FROM GROWING TOO LARGE AND OVERFLOWING.

INSTRUCTOR NOTES

WE PREFER TO HAVE Rational BE A PRIVATE TYPE RATHER THAN A LIMITED PRIVATE TYPE SO THAT WE MAY USE := INSTEAD OF A PROCEDURE Assign FOR ASSIGNMENT. THUS, THE OPERATIONS OF TYPE Rational LOOK JUST LIKE THOSE OF THE OTHER Ada NUMERIC TYPES.

FUNCTION G_C_D IS MORE FULLY DESCRIBED ON THE NEXT SLIDE.

GREATEST COMMON DIVISOR

- WE HAVE TWO REASONS FOR NEEDING AN ALGORITHM FOR EXTRACTING THE GREATEST COMMON DIVISOR (I.E., FACTOR) FROM A NUMERATOR AND A DENOMINATOR:

  1. TO PROVIDE A UNIQUE REPRESENTATION FOR EACH ABSTRACT RATIONAL VALUE, SO THAT THE = OPERATOR FOR Rational_Type WILL WORK CORRECTLY.

  2. TO KEEP THE NUMERATOR AND DENOMINATOR FROM BECOMING ANY LARGER THAN NECESSARY (AND THEREBY AVOID UNNECESSARY Numeric_Error'S).

- HENCE, OUR IMPLEMENTATION OF Rational_Type WILL MAINTAIN THE FOLLOWING INVARIANT:

```
type Rational_Type is
   record
      Numerator_Part   : Integer;      -- Numerator.
      Denominator_Part : Positive;     -- Denominator.
   end record;

  -- INVARIANT: FOR ALL R IN RATIONAL => G_C_D (R.Numerator_Part,
      R.Denominator_Part) = 1;
```

WHERE G_C_D IS A FUNCTION THAT YIELDS THE GREATEST COMMON DIVISOR OF ITS TWO INTEGER ARGUMENTS.

INSTRUCTOR NOTES

THE G.C.D. ALGORITHM IS PRESENTED FOR COMPLETENESS. DO NOT GO OVER IT. THE LOGIC BEHIND G_C_D IS BEYOND THE SCOPE OF THIS COURSE.

EUCLID'S ALGORITHM FOR GREATEST COMMON DIVISOR

IN APPROXIMATELY 300 B.C., THE GREEK MATHEMATICIAN EUCLID DEVELOPED AN ALGORITHM FOR FINDING THE GREATEST COMMON DIVISOR (G_C_D) OF TWO INTEGERS, I.E., THE LARGEST POSITIVE INTEGER THAT EVENLY DIVIDES BOTH OF THE GIVEN INTEGERS:

```
function G_C_D (Numerator : Integer; Denominator : Positive) return Positive is
    X : Positive := Denominator;
    Y : Natural  := abs Numerator;
    Z : Natural;
begin
    -- INVARIANT: X > 0 and Y >= 0 and G_C_D (Y, X) = G_C_D (Numerator, Denominator);
    while Y > 0 loop
       Z := X mod Y;
       X := Y;
       Y := Z;
    end loop;
    return X;
end G_C_D;
```

INSTRUCTOR NOTES

SHOW CLASS THAT THESE FUNCTION BODIES ARE CORRECT.

IN PARTICULAR, NOTE THAT Make_Rational MUST PRESERVE THE INVARIANT FOR TYPE Rational_Type THAT THE Num AND Denom COMPONENTS HAVE A G_C_D OF 1.

CONSTRUCTION AND SELECTION

THE CONSTRUCTION AND SELECTION OPERATIONS ARE STRAIGHTFORWARD:

```
function Make_Rational (Numerator : Integer; Denominator : Positive)
     return Rational_Type is
        GCD_N_D : constant Positive := G_C_D (Numerator, Denominator);
begin
     return (Numerator / GCD_N_D, Denominator / GCD_N_D);
end Make_Rational;

function Numerator (R : Rational_Type) return Integer is
begin
     return R.Numerator;
end Numerator;

function Denominator (R : Rational_Type) return Positive is
begin
     return R.Denominator_Part;
end Denominator;
```

INSTRUCTOR NOTES

SHOW CLASS THAT THIS FUNCTION BODY IS CORRECT.

IN PARTICULAR, NOTE THAT "+" MUST PRESERVE THE INVARIANT FOR TYPE Rational_Type THAT THE Num AND Denom COMPONENTS HAVE A G_C_D OF 1.

ASK THE CLASS WHAT HAS TO CHANGE TO IMPLEMENT "-".

[THIS MAY STILL HAVE TO COMPUTE INTERMEDIATE VALUES WHICH MAY RAISE Numeric_Error.
 COMMON FACTORS COULD BE REMOVED EARLIER AS FOLLOWS:

 CONSIDER THE FOLLOWING EXAMPLE:

```
1   1      1       1          5            3            8
-   --  = ----- + ----- = --------- + --------- = ---------
6   10    2 * 3   2 * 5   2 * 3 * 5   2 * 5 * 3   2 * 3 * 5

     2 * 2 * 2   2 * 2    4
   = --------- = ----- = -- ]
     2 * 3 * 5   3 * 5   15
```

THE + OPERATOR

```
function "+" (Left, Right : Rational_Type) return Rational_Type is

   Numerator     : constant Integer  := Left.Numerator * Right.Denominator +
                       Right.Numerator * Left.Denominator;
   Denominator   : constant Positive := Left.Denominator * Right.Denominator;
   GCD_N_D : constant Positive := G_C_D (Numerator, Denominator);

begin -- "+"

   return (Numerator / GCD_N_D, Denominator / GCD_N_D);

end "+";
```

```
function "*" (Left, Right : Rational_Type) return Rational_Type is
   Numerator   : Natural;
   Denominator : Positive;
   GCD_N_D     : Positive;
begin -- "*"
   Numerator   := Left.Numerator_Part * Right.Numerator_Part;
   Denominator := Left.Denominator_Part * Right.Denominator_Part;
   GCD_N_D     := G_C_D_ (Numerator, Denominator);
   return (Numerator/GCD_N_D, Denominator/GCD_N_D);
end "*";
```

EXERCISE

WRITE THE FUNCTION BODY FOR "*"

HINT: $\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd}$

INSTRUCTOR NOTES

SHOW CLASS THAT THESE FUNCTION BODIES ARE CORRECT.

IN PARTICULAR, NOTE THAT ALTHOUGH INVERSE AND "/" MUST PRESERVE THE INVARIANT FOR TYPE Rational_Type THAT THE Numerator_Part AND Denominator_Part COMPONENTS HAVE A G_C_D OF 1, THEY NEED DO NOTHING SPECIAL TO DO SO.

NOTE THAT THIS VERSION OF "/" USES THE * OPERATOR FOR TYPE Rational_Type, NOT THE * OPERATOR FOR TYPE Integer.

DIVISION OPERATOR

A VERSION OF THE / OPERATOR THAT USES AN AUXILIARY FUNCTION Inverse THAT YIELDS THE RECIPROCAL OF A RATIONAL VALUE:

```
function Inverse (R : Rational_Type) return Rational_Type is
begin
    if R.Numerator_Part > 0 then
         return (R.Denominator_Part, R.Numerator_Part);
    elsif R.Numerator_Part < 0 then
         return (-R.Denominator_Part, -R.Numerator_Part);
    else
         raise Numeric_Error;
    end if;
end Inverse;

function "/" (Left, Right : Rational_Type) return Rational_Type is
begin
    return Left * Inverse (Right);     -- RATIONAL "*".
end "/";
```

INSTRUCTOR NOTES

SHOW CLASS THAT THIS FUNCTION BODY IS CORRECT.

IN PARTICULAR, NOTE THAT ALTHOUGH "**" MUST PRESERVE THE INVARIANT FOR TYPE Rational_Type THAT THE Num AND Denom COMPONENTS HAVE A G_C_D OF 1, IT IS ALREADY TRUE SINCE ** IS EQUIVALENT TO REPEATED *.

A STRAIGHTFORWARD VERSION OF THE ** OPERATOR:

```
function "**" (Left : Rational_Type; Right : Integer) return Rational_Type is
   Power : Rational_Type := (Left.Numerator_Part   ** (abs Right),
                             Left.Denominator_Part ** (abs Right));
begin
   if Right > 0 then
      return Power;
   else
      return Inverse (Power);
   end if;
end "**";
```

INSTRUCTOR NOTES

SHOW CLASS THAT THIS FUNCTION BODY IS CORRECT.

$$L < R$$

$$\frac{L.N}{L.D} < \frac{R.N}{R.D}$$

$$\frac{L.N}{L.D} * L.D * R.D < \frac{R.N}{R.D} * L.D * R.D$$

$$L.N * R.D < R.N * L.D$$

THE < OPERATOR:

```
function "<" (Left, Right : Rational_Type) return Boolean is
begin
    return Left.Numerator_Part * Right.Denominator_Part < Right.Numerator_Part *
      Left.Denominator_Part;
end "<";
```

INSTRUCTOR NOTES

SECTION 11

GENERIC UNITS

INSTRUCTOR NOTES

THE PURPOSE OF EACH OF THESE PROCEDURES IS TO EXCHANGE THE VALUES OF ITS TWO PARAMETERS.

WE CAN'T WRITE ONE PROCEDURE BECAUSE STRONG TYPING REQUIRES EACH PROCEDURE TO SPECIFY THE ONE TYPE OF EACH OF ITS PARAMETERS.

THE NEED FOR GENERIC UNITS

SUPPOSE WE WISH TO IMPLEMENT SIMILAR OPERATIONS FOR SEVERAL DIFFERENT TYPES.

AT FIRST GLANCE, STRONG TYPING SEEMS TO REQUIRE DUPLICATE CODING:

```
procedure Swap_Integers (A, B : in out Integer) is
   Old_A : constant Integer := A;
begin
   A := B;
   B := Old_A;
end Swap_Integers;

procedure Swap_Floats (A, B : in out Float) is
   Old_A : constant Float := A;
begin
   A := B;
   B := Old_A;
end Swap_Floats;

procedure Swap_Characters (A, B : in out Character) is
   Old_A : constant Character := A;
begin
   A := B;
   B := Old_A;
end Swap_Characters;
```

INSTRUCTOR NOTES

▨ AND ░ REPRESENT BLANKS. IN A GIVEN INSTANTIATION, BOTH OCCURRENCES OF ▨ ARE FILLED IN THE SAME WAY, AND LIKEWISE FOR ░.

TEMPLATES AND INSTANCES

GENERIC UNITS ARE TEMPLATES FOR SUBPROGRAMS OR PACKAGES, WITH CERTAIN ITEMS LEFT BLANK.

AN INSTANCE OF THE TEMPLATE CAN BE CREATED BY FILLING IN THE BLANKS. THIS IS CALLED INSTANTIATION.

TEMPLATE:

```
procedure ▨ (A, B : in out ░ ) is
   Old_A : constant ░ := A;
begin
   A := B;
   B := Old_A;
end ▨ ;
```

INSTANTIATIONS ON PREVIOUS SLIDE:

1. ▨ = Swap_Integers, ░ = Integer
2. ▨ = Swap_Floats, ░ = Float
3. ▨ = Swap_Characters, ░ = Character

INSTRUCTOR NOTES

MAKE SURE THE CLASS UNDERSTANDS THE DISTINCTION BETWEEN TEMPLATES AND INSTANCES OF TEMPLATES BEFORE PROCEEDING.

with CLAUSES APPLY BOTH TO SEPARATELY COMPILED TEMPLATES AND SEPARATELY COMPILED INSTANCES.

(THERE IS NO SUCH THING AS A GENERIC TASK UNIT.)

GENERIC UNITS

A GENERIC SUBPROGRAM IS NOT REALLY A SUBPROGRAM, BUT A TEMPLATE FOR SUBPROGRAMS.

- YOU CAN'T CALL A GENERIC SUBPROGRAM, BUT YOU CAN CALL INSTANCES OF IT.

A GENERIC PACKAGE IS NOT REALLY A PACKAGE, BUT A TEMPLATE FOR PACKAGES.

- YOU CAN'T REFER TO ENTITIES PROVIDED BY A GENERIC PACKAGE, BUT YOU CAN REFER TO THE ENTITIES PROVIDED BY INSTANCES OF GENERIC PACKAGES.
- NAMES OF GENERIC PACKAGES CANNOT APPEAR IN use CLAUSES, BUT NAMES OF INSTANCES CAN.

GENERIC SUBPROGRAMS AND GENERIC PACKAGES CAN BE SEPARATELY COMPILED.

- NAMES OF SEPARATELY COMPILED GENERIC SUBPROGRAMS AND GENERIC PACKAGES CAN APPEAR IN A with CLAUSE OF ANOTHER COMPILATION UNIT.
- THIS ALLOWS THE OTHER COMPILATION UNIT TO CONTAIN INSTANTIATIONS OF THE TEMPLATE.

INSTRUCTOR NOTES

AVOID DISCUSSIONS ABOUT SYNTAX AND MECHANICS, WHICH ARE DISCUSSED STARTING WITH THE NEXT SLIDES.

GENERIC TEMPLATE

THE TEMPLATE MAY CONTAIN "BLANKS" CORRESPONDING TO:

- THE NAME OF THE INSTANCE (ALWAYS)
- VALUES
- VARIABLES
- SUBPROGRAMS CALLED FROM WITHIN THE TEMPLATE
- TYPES AND SUBTYPES

INSTRUCTOR NOTES

THE SYNTAX OF GENERIC FORMAL PARAMETERS IS COMPLICATED. IT WILL BE GIVEN LATER.

THE SYNTAX ALLOWS GENERIC TEMPLATES WITH NO PARAMETERS. THIS IS ONLY USEFUL FOR GENERIC PACKAGES CONTAINING VARIABLE DECLARATIONS. EACH INSTANCE OF THE TEMPLATE IS AN IDENTICAL COPY OF THE PACKAGE WITH ITS OWN DISTINCT SET OF VARIABLES.

FORM OF A GENERIC UNIT

```
generic
  {generic formal parameter}  )
unit declaration              )    generic declaration

unit body                     }    generic body
```

unit declaration IS EITHER A SUBPROGRAM DECLARATION OR A PACKAGE DECLARATION.

unit body IS EITHER A SUBPROGRAM BODY OR A PACKAGE BODY, RESPECTIVELY.

A generic formal parameter DECLARES CERTAIN NAMES TO STAND FOR "BLANKS" IN THE UNIT DECLARATION AND UNIT BODY

UNLIKE AN ORDINARY SUBPROGRAM, A GENERIC SUBPROGRAM MUST ALWAYS HAVE BOTH A DECLARATION AND A BODY.

INSTRUCTOR NOTES

THIS IS THE FIRST GENERIC FORMAL PARAMETER THE CLASS HAS SEEN, AND THEY DON'T YET UNDERSTAND WHAT IT MEANS.

EXPLAIN THAT WHAT LOOKS LIKE A PRIVATE TYPE DECLARATION FOLLOWING THE WORD generic IS REALLY A GENERIC PARAMETER DECLARATION.

IT STATES THAT Parameter_Type STANDS FOR A TYPE HAVING ALL THE OPERATIONS AVAILABLE FOR A PRIVATE TYPE, SUCH AS ASSIGNMENT. ONLY OPERATIONS AVAILABLE FOR PRIVATE TYPES MAY BE USED WITHIN THE GENERIC BODY. ("A := A + B;" WOULD BE ILLEGAL.) THE "BLANK" NAMED Parameter_Type MAY BE FILLED IN WITH ANY TYPE HAVING THESE OPERATIONS (I.E., ANY TYPE BUT A LIMITED PRIVATE TYPE) TO CREATE AN INSTANCE.

ASSURE THE CLASS THAT GENERIC FORMAL PARAMETERS WILL BE EXPLAINED IN DETAIL SHORTLY.

EXAMPLE OF A GENERIC PROCEDURE

```
generic
   type Parameter_Type is private;
procedure Swap_Template (A, B : in out Parameter_Type);

procedure Swap_Template (A, B : in out Parameter_Type) is
   Old_A : constant Parameter_Type := A;
begin -- Swap_Template
   A := B;
   B := Old_A;
end Swap_Template;
```

- Parameter_Type ACTS AS A BLANK FOR A TYPE NAME.

- Swap_Template ACTS AS A BLANK FOR A PROCEDURE NAME.

INSTRUCTOR NOTES

THIS SLIDE RE-EMPHASIZES A POINT MADE ON AN EARLIER SLIDE. ASK THE CLASS HOW TO CORRECT THE ERROR.

REMEMBER, THE FOLLOWING IS ILLEGAL:

```
generic
   type Parameter_Type is private;
procedure Swap_Template (A, B : in out Parameter_Type) is
   Old_A : constant Parameter_Type := A;
begin -- Swap_Template
   A := B;
   B := Old_A;
end Swap_Template;
```

A GENERIC PROCEDURE MUST HAVE BOTH A DECLARATION AND A BODY, AND THE GENERIC FORMAL PARAMETERS GO WITH THE DECLARATION.

INSTRUCTOR NOTES

EXPLAIN THAT THE SECOND LINE OF THE GENERIC DECLARATION DECLARES Object_Type TO BE A GENERIC FORMAL PARAMETER STANDING FOR A DISCRETE TYPE.

IF X IS A VALUE IN THE DISCRETE TYPE, THE FUNCTION CALL Current_Count (X) RETURNS THE NUMBER OF TIMES Count_Object HAS BEEN CALLED WITH PARAMETER X.

QUICKLY WALK THROUGH THE PACKAGE BODY.

EXAMPLE OF A GENERIC PACKAGE

```
generic
   type Object_Type is (< >);
package Counting_Package_Template is
   procedure Count_Object (Object : in Object_Type);
   function Current_Count (Object : Object_Type) return Positive;
end Counting_Package_Template;

package body Counting_Package_Template is

   Count_Table : array (Object_Type) of Natural := (Object_Type =>  0);

   procedure Count_Object (Object : in Object_Type) is
   begin -- Count_Object
      Count_Table (Object) := Count_Table (Object) + 1;
   end Count_Object;

   function Current_Count (Object : Object_Type) return Positive is
   begin -- Current_Count
      return Count_Table (Object);
   end Count_Object;

end Counting_Package_Template;
```

- Object_Type ACTS AS A BLANK FOR THE NAME OF A DISCRETE TYPE.

- Counting_Package_Template ACTS AS A BLANK FOR A PACKAGE NAME.

INSTRUCTOR NOTES

FOR A WHILE, WE WILL SHOW ONLY NAMED ACTUAL PARAMETERS IN GENERIC INSTANTIATIONS. THERE IS NO REASON TO LET ON AT THIS POINT THAT THERE ARE OTHER FORMS.

EMPHASIZE THAT WRITING AN INSTANTIATION IS EQUIVALENT TO WRITING AN INSTANCE OF THE GENERIC DECLARATION AT THAT PLACE.

NO FORMAL DESCRIPTION OF THE SYNTAX FOLLOWS. THE CLASS'S UNDERSTANDING OF THE STRUCTURE OF A SUBPROGRAM INSTANTIATION MUST COME FROM THIS AND SUBSEQUENT EXAMPLES.

EXPLAIN THAT A GENERIC FUNCTION INSTANTIATION IS THE SAME EXCEPT FOR THE FIRST WORD.

EXAMPLE OF A GENERIC SUBPROGRAM INSTANTIATION

```
procedure Swap_Integers is new
   Swap_Template (Parameter_Type =>  Integer);
```

- CREATES AN INSTANCE OF Swap_Template WITH Swap_Template REPLACED BY Swap_Integers AND Parameter_Type REPLACED BY Integer.

- THE TYPE Integer IS A GENERIC ACTUAL PARAMETER CORRESPONDING TO THE GENERIC FORMAL PARAMETER Parameter_Type.

- EQUIVALENT TO GETTING SPECIFICATION AT THE PLACE OF THE INSTANTIATION AND THE BODY SOMEWHERE LATER.

```
procedure Swap_Integers (A, B : in out Integer);

procedure Swap_Integers (A, B : in out Integer) is
   Old_A : constant Integer := A;
begin
   A := B;
   B := Old_A;
end Swap_Integers;
```

INSTRUCTOR NOTES

NO FORMAL DESCRIPTION OF THE SYNTAX FOLLOWS. THE CLASS'S UNDERSTANDING OF THE STRUCTURE OF A PACKAGE INSTANTIATION MUST COME FROM THIS AND SUBSEQUENT EXAMPLES.

EXAMPLE OF A GENERIC PACKAGE INSTANTIATION

```
package Character_Counting_Package is new
   Counting_Package_Template (Object_Type =>  Character);
```

- CREATES AN INSTANCE OF Counting_Package_Template WITH Counting_Package_Template

  REPLACED BY Character_Counting_Package AND Object_Type REPLACED BY Character.

- EQUIVALENT TO WRITING THE FOLLOWING IN PLACE OF THE INSTANTIATION:

```
package Character_Counting_Package is
   procedure Count_Object (Object : in Character);
   function Current_Count (Object: in Character) return Positive;
end Character_Counting_Package;

package body Character_Counting_Package is

   Count_Table : array (Character) of Natural := (Character =>  0);

   procedure Count_Object (Object : in Character) is
   begin -- Count_Object
      Count_Table (Object) := Count_Table (Object) + 1;
   end Count_Object;

   function Current_Count (Object : Character) return Positive is
   begin -- Current_Count
      return Count_Table (Object);
   end Current_Count;

end Character_Counting_Package;
```

INSTRUCTOR NOTES

THIS SLIDE SHOWS HOW GENERIC UNITS SOLVE THE PROBLEM PRESENTED AT THE BEGINNING OF THE SECTION.

SINCE A GENERIC PROCEDURE INSTANTIATION NAMED Swap IS EQUIVALENT TO A PROCEDURE DECLARATION AND PROCEDURE BODY WITH THAT NAME, THE THREE LOWER INSTANTIATIONS OVERLOAD THREE VERSIONS OF Swap. AN INSTANTIATION MAY OVERLOAD ANOTHER INSTANTIATION, AN ORDINARY SUBPROGRAM, OR A PREDEFINED SUBPROGRAM.

MULTIPLE INSTANTIATIONS OF THE SAME TEMPLATE

TO DECLARE THE THREE ALMOST-IDENTICAL PROCEDURES SHOWN EARLIER:

```
procedure Swap_Integers is new
   Swap_Template (Parameter_Type =>  Integer);

procedure Swap_Floats is new
   Swap_Template (Parameter_Type =>  Float);

procedure Swap_Characters is new
   Swap_Template (Parameter_Type =>  Character);
```

ALTERNATIVELY, THE FOLLOWING INSTANTIATIONS PRODUCE THREE OVERLOADED PROCEDURES NAMED Swap:

```
procedure Swap is new
   Swap_Template (Parameter_Type =>  Integer);

procedure Swap is new
   Swap_Template (Parameter_Type =>  Float);

procedure Swap is new
   Swap_Template (Parameter_Type =>  Character);
```

INSTRUCTOR NOTES

WHEN A GENERIC UNIT APPEARS IN A DECLARATIVE PART, THE INSTANTIATION MUST APPEAR IN THE SAME DECLARATIVE PART, A NESTED DECLARATIVE PART, OR A DECLARATIVE PART NESTED IN THE CORRESPONDING SEQUENCE OF STATEMENTS. IN THE FIRST CASE, THE INSTANTIATION MAY APPEAR ANY TIME AFTER THE GENERIC DECLARATION, EVEN BEFORE THE GENERIC BODY.

THE PLACEMENT OF A GENERIC UNIT PROVIDED BY A PACKAGE IS EXPLAINED AS FOLLOWS: THE GENERIC DECLARATION DESCRIBES HOW TO INSTANTIATE THE TEMPLATE AND HOW TO USE THE INSTANTIATION. THE GENERIC BODY DESCRIBES HOW THE FACILITIES PROVIDED BY AN INSTANTIATION ARE IMPLEMENTED.

WHEN THE GENERIC DECLARATION AND GENERIC BODY ARE COMPILATION UNITS, AN INSTANTIATION MAY BE COMPILED ANY TIME AFTER THE GENERIC DECLARATION. THE ADA REFERENCE MANUAL GIVES AN IMPLEMENTATION THE OPTION OF IMPOSING THE SEPARATE COMPILATION REQUIREMENT.

WHERE DO GENERIC UNITS GO?

- IN A DECLARATIVE PART OF A BLOCK STATEMENT, SUBPROGRAM BODY, PACKAGE BODY, OR TASK BODY.

  -- FIRST THE GENERIC DECLARATION

  -- THE GENERIC BODY FOLLOWS SOME TIME LATER

- PROVIDED BY A PACKAGE

  -- GENERIC DECLARATION IN PACKAGE DECLARATION

  -- GENERIC BODY IN PACKAGE BODY

- SEPARATELY COMPILED

-- THE GENERIC DECLARATION AND GENERIC BODY ARE TWO DISTINCT COMPILATION UNITS, BUT SOME IMPLEMENTATIONS MAY REQUIRE THAT THEY APPEAR IN THE SAME COMPILATION.

INSTRUCTOR NOTES

WE LIED EARLIER. WHEN AN INSTANTIATION OCCURS IN A PACKAGE SPECIFICATION, IT IS NOT EQUIVALENT TO THE DECLARATION AND BODY OF THE INSTANCE APPEARING AT THAT POINT (WHICH WOULD BE ILLEGAL), BUT TO THE DECLARATION OF THE INSTANCE APPEARING THERE AND THE BODY OF THE INSTANCE APPEARING IN THE PACKAGE BODY. IT MAY BE BEST TO IGNORE THIS POINT IF A STUDENT DOES NOT BRING IT UP.

A SEPARATELY COMPILED PROCEDURE/FUNCTION/PACKAGE INSTANTIATION IS EQUIVALENT TO A SEPARATELY COMPILED PROCEDURE/FUNCTION/PACKAGE. THE INSTANCE MAY BE NAMED IN A with CLAUSE.

WHERE DO GENERIC INSTANTIATIONS GO?

ANY PLACE A SUBPROGRAM DECLARATION OR PACKAGE DECLARATION CAN GO:

- IN A DECLARATIVE PART OF A BLOCK STATEMENT, SUBPROGRAM BODY, PACKAGE BODY, OR TASK BODY

- IN A PACKAGE SPECIFICATION

- AS A SEPARATE COMPILATION UNIT

INSTRUCTOR NOTES

EACH KIND OF FORMAL PARAMETER HAS A DIFFERENT FORM, AND EACH FORM HAS TWO OR MORE VARIATIONS.

GENERIC FORMAL OBJECTS, GENERIC FORMAL SUBPROGRAMS, AND GENERIC FORMAL TYPES WILL BE CONSIDERED IN TURN.

THIS IS AN APPROPRIATE BREAK POINT IF NEEDED.

GENERIC FORMAL PARAMETERS

- GENERIC FORMAL OBJECTS

  STAND FOR VALUES AND VARIABLES.

- GENERIC FORMAL SUBPROGRAMS

  STAND FOR PROCEDURES AND FUNCTIONS WITH SPECIFIED PARAMETER/RESULT TYPES.

- GENERIC FORMAL TYPES

  STAND FOR TYPES OR SUBTYPES.

INSTRUCTOR NOTES

WITHIN A GENERIC DECLARATION OR GENERIC BODY, A GENERIC FORMAL CONSTANT MAY BE USED AS AN ORDINARY CONSTANT AND A GENERIC FORMAL VARIABLE MAY BE USED AS AN ORDINARY VARIABLE.

THE GENERIC ACTUAL PARAMETER CORRESPONDING TO A GENERIC FORMAL VARIABLE MUST BE A VARIABLE (POSSIBLY AN ARRAY COMPONENT, RECORD COMPONENT, OR ALLOCATED VARIABLE).

GENERIC FORMAL OBJECTS

- <u>GENERIC FORMAL CONSTANTS</u>:

  ```
  identifier {, identifier}  : in type or subtype name ;
  ```

  THE DECLARED IDENTIFIERS STAND FOR THE VALUES OF THE CORRESPONDING GENERIC ACTUAL PARAMETERS

- <u>GENERIC FORMAL VARIABLES</u>:

  ```
  identifier {, identifier} : in out type or subtype name ;
  ```

  THE DECLARED IDENTIFIERS STAND FOR THE VARIABLES SPECIFIED AS GENERIC ACTUAL PARAMETERS. THE GENERIC FORMAL PARAMETERS CAN BE USED IN ANY WAY THOSE VARIABLES CAN BE USED.

- THERE ARE NO GENERIC FORMAL OBJECTS OF MODE <u>out</u>.

- THE WORD <u>in</u> MAY BE OMITTED FOR GENERIC FORMAL CONSTANTS, BUT WE DON'T RECOMMEND IT.

INSTRUCTOR NOTES

POINT OUT THE DISTINCTION BETWEEN THE SUBPROGRAM FORMAL PARAMETER X AND THE GENERIC FORMAL PARAMETER Increment.

EXAMPLE OF A GENERIC FORMAL CONSTANT

GENERIC UNIT:

```
generic
   Increment : in Integer;                         generic formal parameter
procedure Add_Increment (X: in out Integer);

procedure Add_Increment (X : in out Integer) is     subprogram formal parameter
begin
   X := X + Increment;
end Add_Increment;
```

INSTANTIATIONS:

```
procedure Main is
   A, B : Integer;
   Step : Positive := 5;
      ...
   procedure Add_Step is
      new Add_Increment (Increment =>  Step);
                                                     generic actual parameter
   procedure Add_5 is
      new Add_Increment (Increment =>  5);

begin -- Main
   ...
   Add_Step (X =>  A);
   ...                                      subprogram actual parameter
   Add_5 (X =>  B);
end Main;
```

INSTRUCTOR NOTES

THE FIRST INSTANTIATION IS EQUIVALENT TO A PROCEDURE WHOSE BODY REFERENCES A AS A GLOBAL VARIABLE. THE SECOND INSTANTIATION IS EQUIVALENT TO A PROCEDURE WHOSE BODY REFERENCES B AS A GLOBAL VARIABLE.

THIS EXAMPLE DOES NOT REFLECT RECOMMENDED PROGRAMMING PRACTICE.

EXAMPLE OF A GENERIC FORMAL VARIABLE

<u>GENERIC UNIT:</u>

```
generic
    Target : in out Integer;
procedure Double_Target;

procedure Double_Target is
begin
   Target := 2 * Target;
end Double_Target;
```

<u>INSTANTIATIONS:</u>

```
procedure Main is

   A : Integer;
   B : Integer range 0 .. 1000;
   ...
   procedure Double_A is new Double_Target (Target =>  A);
   procedure Double_B is new Double_Target (Target =>  B);

begin
   ...
   Double_A;
   ...
   Double_B;
   ...
end Main;
```

- RANGE CONSTRAINT ON B APPLIES DURING CALLS ON Double_B.

INSTRUCTOR NOTES

THESE TWO GENERIC UNITS ARE IDENTICAL EXCEPT FOR THEIR NAMES AND THE MODES OF THE GENERIC FORMAL OBJECTS.

THE DISTINCTIONS MADE IN THE COMMENTS ARE ILLUSTRATED BY INSTANTIATIONS ON THE NEXT SLIDE.

DISTINCTION BETWEEN GENERIC FORMAL CONSTANTS AND GENERIC FORMAL VARIABLES

```
generic
   Increment : in Integer;    -- GENERIC FORMAL CONSTANT DECLARATION
procedure Add_Fixed_Increment (N : in out Integer);

procedure Add_Fixed_Increment (N : in out Integer) is
begin
   N := N + Increment;        -- Increment IS A CONSTANT
end Add_Fixed_Increment;

          -- VALUE OF Increment IS FIXED AT TIME OF INSTANTIATION, WHEN
          -- THE GENERIC ACTUAL PARAMETER IS EVALUATED

generic
   Increment : in out Integer;     -- GENERIC FORMAL VARIABLE DECLARATION
procedure Add_Current_Increment (N : in out Integer);

procedure Add_Current_Increment (N : in out Integer) is
begin
   N := N + Increment;   -- Increment STANDS FOR A GLOBAL VARIABLE
end Add_Current_Increment;

          -- VALUE OF increment DURING ANY CALL ON THE PROCEDURE IS THE CURRENT
          -- VALUE OF THE VARIABLE SPECIFIED AS A GENERIC ACTUAL PARAMETER
```

INSTRUCTOR NOTES

THE FIRST INSTANTIATION BINDS Increment TO THE CURRENT VALUE OF X (NAMELY 10). THE SECOND INSTANTIATION BINDS Increment TO THE VARIABLE X, SO THAT THE PROCEDURE Add_X IMPLICITLY REFERS TO X AS A GLOBAL VARIABLE.

CHANGING THE VALUE OF X AFFECTS THE BEHAVIOR OF Add_X BUT NOT OF Add_10.

DISTINCTION BETWEEN GENERIC FORMAL CONSTANTS AND GENERIC FORMAL VARIABLES (Continued)

```
procedure Main is

   X : Integer := 10;
   Y : Integer := 0;

   procedure Add_10 is new Add_Fixed_Increment (Increment =>  X);
       -- EQUIVALENT TO (Increment => 10)
   procedure Add_X is new Add_Current_Increment (Increment =>  X);

begin -- Main

   Add_10 (Y);          -- INCREASES Y BY 10
   Add_X (Y);           -- ALSO INCREASES Y BY 10
   X := 5;
   Add_10 (Y);          -- STILL INCREASES Y BY 10
   Add_X (Y);           -- NOW INCREASES Y BY 5

end Main;
```

INSTRUCTOR NOTES

```
generic
   Variable : in out Integer;
   To       : in Integer;
procedure Reset_Variable;

procedure Reset_Variable is
begin
   Variable := To;
end Reset_Variable;
```

EXERCISE

WRITE A GENERIC PROCEDURE Reset_Variable SUCH THAT AFTER THE INSTANTIATION

```
PROCEDURE Reset_X_To_10 is new Reset_Variable (Variable => X, To => 10);
```

THE PROCEDURE CALL

```
Reset_X_To_10;
```

WILL PLACE THE VALUE 10 IN X.

THE GENERIC PARAMETERS SPECIFY THE VARIABLE TO BE "RESET" AND THE VALUE TO WHICH IT SHOULD BE RESET. BOTH ARE OF TYPE Integer.

INSTRUCTOR NOTES

IF NECESSARY, REVIEW THE TWO FORMS OF SUBPROGRAM SPECIFICATIONS:

```
procedure identifier [(parameter specification {; parameter specification })]

function designator [(parameter specification {; parameter specification })]
   return type or subtype name
```

(A DESIGNATOR IS EITHER AN IDENTIFIER OR AN OPERATOR SYMBOL LIKE "+".)

GENERIC FORMAL SUBPROGRAMS

```
with [subprogram specification] ;
```

- THE SUBPROGRAM NAME GIVEN IN THE SUBPROGRAM SPECIFICATION STANDS FOR THE SUBPROGRAM SPECIFIED AS A GENERIC ACTUAL PARAMETER.

- THE FORMAL AND ACTUAL SUBPROGRAMS MUST BE:

  - EITHER TWO PROCEDURES WITH MATCHING PARAMETER TYPES AND MODES

  - OR TWO FUNCTIONS WITH MATCHING PARAMETER AND RESULT TYPES

- THE GENERIC FORMAL SUBPROGRAM MAY BE CALLED FROM WITHIN THE GENERIC UNIT IN ACCORDANCE WITH ITS SUBPROGRAM SPECIFICATION.

INSTRUCTOR NOTES

Day_Type IS AN ENUMERATION TYPE.

Day_Set_Type IS A TYPE FOR SETS OF Day_Type VALUES, IMPLEMENTED USING AN ARRAY OF BOOLEANS AS DESCRIBED EARLIER IN SECTION II.A.

Reset_Hours IS A PROCEDURE TAKING A Day_Type PARAMETER AND SETTING THE CORRESPONDING ELEMENT OF THE ARRAY Hours TO ZERO.

Day_After IS A FUNCTION TAKING A Day_Type PARAMETER AND RETURNING THE VALUE CORRESPONDING TO THE NEXT DAY, EVEN WHEN GIVEN Day_Type'Last AS A PARAMETER.

THESE ENTITIES ARE USED IN EXAMPLES ON THE FOLLOWING SLIDES.

EXAMPLES OF GENERIC FORMAL SUBPROGRAMS

CONTEXT FOR EXAMPLE 1 AND EXAMPLE 2:

```
type Day_Type is
   (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

type Day_Set_Type is array (Day_Type) of Boolean;

Hours : array (Day_Type) of Integer range 0 .. 24;

procedure Reset_Hours (Day : in Day_Type) is
begin -- Reset_Hours
   Hours (Day) := 0;
end Reset_Hours;

function Day_After (Day : Day_Type) return Day_Type is
begin -- Day_After
   if Day = Day_Type'Last then
      return Day_Type'First;
   else
      return Day_Type'Succ (Day);
   end if;
end Day_After;
```

INSTRUCTOR NOTES

GIVEN A GENERIC ACTUAL PARAMETER CONVEYING WHAT IT MEANS TO "PROCESS" A Day_Type VALUE, AN INSTANTIATION OF Process_Each_Element TAKES A Day_Set_Type PROCEDURE PARAMETER AND "PROCESSES" EACH ELEMENT OF THE SET.

POINT OUT THE GENERIC FORMAL PROCEDURE Process_One_Element AND THE DECLARATION OF THE GENERIC PROCEDURE Process_Each_Element.

Reset_Hours_Of_Day_Set_Elements IS ESSENTIALLY A COPY OF Process_Each_Element, WITH THE CALL ON Process_One_Element REPLACED BY A CALL ON Reset_Hours.

POINT OUT THAT THE DECLARATION OF Process_One_Element MATCHES THE SPECIFICATION OF Reset_Hours ON THE PREVIOUS SLIDE.

EXAMPLES OF GENERIC FORMAL SUBPROGRAMS -- EXAMPLE 1

GENERIC PROCEDURE:

```
generic
   with procedure Process_One_Element (Element : in Day_Type);
procedure Process_Each_Element (Set : in Day_Set_Type);

procedure Process_Each_Element (Set : in Day_Set_Type) is
begin -- Process_Each_Element
   for D in Day_Type loop
      if Set (D) then -- D is in Set
         Process_One_Element (Element => D);
      end if;
   end loop;
end Process_Each_Element;
```

GENERIC FORMAL PROCEDURE

GENERIC PROCEDURE

INSTANTIATION:

```
procedure Reset_Hours_Of_Day_Set_Elements is
   new Process_Each_Element (Process_One_Element =>  Reset_Hours);
```

USE:

IF DS IS A Day_Set_Type VALUE, THE PROCEDURE CALL

```
Reset_Hours_Of_Day_Set_Elements (DS);
```

SETS Hours (D) TO 0 FOR EACH ELEMENT D IN DS.

INSTRUCTOR NOTES

GIVEN THE DEFINITION OF THE "IMAGE" OF A Day_Type VALUE, AN INSTANTIATION OF Set_Image TAKES A Day_Set_Type PARAMETER AND RETURNS THE SET OF IMAGES OF DAYS IN THAT SET.

IF TWO DAYS IN THE ORIGINAL SET HAVE THE SAME IMAGE VALUE, THE RESULTING SET WILL HAVE FEWER ELEMENTS THAN THE ORIGINAL SET. (THIS CANNOT HAPPEN WITH THE IMAGE FUNCTION Day_After.)

DISTINGUISH BETWEEN THE GENERIC FORMAL FUNCTION Element_Image AND THE GENERIC FUNCTION Set_Image.

Set_Of_Successor_Days IS ESSENTIALLY A COPY OF Set_Image WITH THE CALL ON Element_Image REPLACED BY A CALL ON Day_After.

POINT OUT THAT THE SPECIFICATION OF Element_Image IN THE GENERIC FORMAL PARAMETER MATCHES THE DECLARATION OF Day_After TWO SLIDES EARLIER.

EXAMPLES OF GENERIC FORMAL SUBPROGRAMS -- EXAMPLE 2

GENERIC FUNCTION:

```
generic
   with function Element_Image (Element : Day_Type) return Day_Type;
function Set_Image (Set : Day_Set_Type) return Day_Set_Type;

function Set_Image (Set : Day_Set_Type) return Day_Set_Type is
   Result : Day_Set_Type := (Day_Type =>  False);        -- EMPTY SET
begin -- Set_Image
   for D in Day_Type loop
      if Set (D) then                      -- D IS IN SET
         Result (Element_Image (D)) := True;  -- ADD Element_Image (D) TO Result
      end if;
   end loop;
   return Result;
end Set_Image;
```

INSTANTIATION:

```
function Set_Of_Successor_Days is
   new Set_Image (Element_Image =>  Day_After);
```

USE:

IF DS IS A Day_Set_Type VALUE,

Set_Image (DS) = {Element_Image (D) | D ∈ DS}.

IF DS = {Monday, Thursday, Saturday},

Set_Of_Successor_Days (DS) = {Tuesday, Friday, Sunday}.

INSTRUCTOR NOTES

THE REASON FOR THE EQUALITY IS THAT EACH $F(X_i)$ OTHER THAN $F(X_o)$ AND $F(X_n)$ OCCURS IN TWO SUCCESSIVE TERMS OF THE LEFTHAND SUM, DIVIDED BY 2 IN EACH CASE.

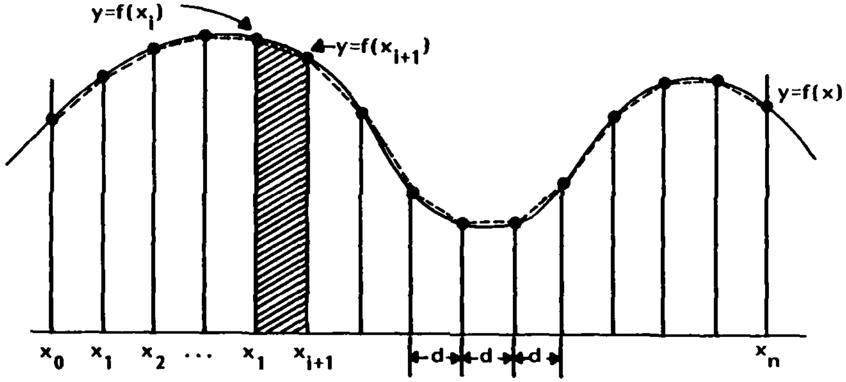
CONCENTRATE ON THE FACT THAT THE ACTUAL FUNCTION IS TO BE PASSED AS AN ACTUAL PARAMETER TO THE GENERIC. DON'T CONCENTRATE ON THE TRAPEZOIDAL RULE, THIS IS NOT A COURSE IN CALCULUS.

EXAMPLES OF GENERIC FORMAL SUBPROGRAMS

BACKGROUND FOR EXAMPLE 3:

NUMERIC INTEGRATION IS THE APPROXIMATE COMPUTATION OF THE AREA UNDER A CURVE BETWEEN TWO POINTS.

THE TRAPEZOIDAL METHOD DIVIDES THE X-AXIS BETWEEN THESE POINTS INTO EVENLY-SPACED INTERVALS AND APPROXIMATES THE CURVE WITHIN AN INTERVAL BY A STRAIGHT LINE:



THIS DIVIDES THE AREA UNDER THE CURVE INTO A SEQUENCE OF TRAPEZOIDS. *IF d IS THE DISTANCE BETWEEN POINTS ON THE X-AXIS,* THE AREA OF THE TRAPEZOID BETWEEN POINTS x_i AND x_{i+1} IS $d \bullet (f(x_i) + f(x_{i+1})) / 2$.

THE APPROXIMATE AREA UNDER THE CURVE BETWEEN x_0 AND x_n IS:

$$\sum_{i=0}^{n-1} d \bullet (f(x_i) + f(x_{i+1})) / 2 = d \bullet \left(\frac{f(x_0) + f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i) \right)$$

INSTRUCTOR NOTES

THE ACTUAL PARAMETER TO Curve INSIDE THE for LOOP CORRESPONDS TO THE VALUE X_i ON THE PREVIOUS SLIDE, AND Interval CORRESPONDS TO d.

THIS ALGORITHM ALSO WORKS FOR TO ≤ FROM. CONVENTIONALLY, THE DEFINITE INTEGRAL FROM _a_ TO _b_ IS THE NEGATION OF THE DEFINITE INTEGRAL FROM _b_ TO _a_.

THE EXAMPLE COMPUTES THE AREA UNDER A BELL CURVE BETWEEN TWO GIVEN POINTS. THIS AREA IS THE PROBABILITY OF A NORMALLY DISTRIBUTED RANDOM VARIABLE FALLING BETWEEN THOSE TWO POINTS.

EXAMPLES OF GENERIC FORMAL SUBPROGRAMS -- EXAMPLE 3

GENERIC FUNCTION:

```
generic
   with function Curve (X : Float) return Float;
function Area_Under_Curve
   (From, To : Float; Number_Of_Trapezoids : Positive) return Float;

function Area_Under_Curve
   (From, To : Float; Number_Of_Trapezoids : Positive) return Float is

   Interval, Sum : Float;

begin -- Area_Under_Curve

   Interval := (To - From) / Float (Number_Of_Trapezoids);
   Sum := (Curve (From) + Curve (To)) / 2.0;
   for I in 1 .. Number_Of_Trapezoids - 1 loop
      Sum := Sum + Curve (From + Float (I) * Interval);
   end loop;
   return Interval * Sum;

end Area_Under_Curve;
```

INSTANTIATION:

```
with Math_Package;
function Normal_Curve (X : Float) return Float is
begin
   return Math_Package.Exp (- X ** 2);    -- e ** (-(X ** 2))
end Normal_Curve;

function Normal_Probability_Distribution
   is new Area_Under_Curve (Curve =>  Normal_Curve);
```

USE:

```
Within_1_Standard_Deviation := Normal_Probability_Distribution (-1.0, 1.0, 1000);
```

```
generic
   with Repeated_Function (X : Float) return Float;
   N : in Positive;
function Apply_N_Times (X : Float) return Float;

function Apply_N_Times (X : Float) return Float is
   Result : Float := X;
begin
   for I in 1 .. N loop
      Result := Repeated_Function (Result);
   end loop;
   return Result;
end Apply_N_Times;
```

EXERCISE

WRITE A GENERIC FUNCTION Apply_N_Times SUCH THAT AFTER THE INSTANTIATION

```
      function Square_3_Times is new Apply_N_Times (Repeated_Function => Square, N => 3);
```

THE FUNCTION CALL

```
      Square_3_Times (X)
```

RETURNS THE VALUE

```
      Square (Square (Square (X))).
```

THE FIRST GENERIC PARAMETER GIVES A FUNCTION TO BE REPEATEDLY APPLIED AND THE SECOND GIVES A POSITIVE VALUE SPECIFYING HOW MANY TIMES THE FUNCTION IS TO BE APPLIED. THE FUNCTION TO BE REPEATEDLY APPLIED SHOULD TAKE A SINGLE PARAMETER OF TYPE Float AND RETURN A RESULT OF TYPE Float.

INSTRUCTOR NOTES

DON'T GO INTO DETAILS. ALL CLASSES OF GENERIC FORMAL TYPES ARE COVERED IN DETAIL LATER.

THE PURPOSE OF THIS SLIDE IS TO ILLUSTRATE WHAT GENERIC FORMAL TYPE DECLARATIONS LOOK LIKE:

1. THEY LOOK ROUGHLY LIKE TYPE DECLARATIONS.

2. BOXES, < > , ARE USED IN SOME CASES TO INDICATE MISSING INFORMATION (A LIST OF ENUMERATION LITERALS, A RANGE, AN INTEGER VALUE, AND A REAL VALUE, RESPECTIVELY). REMIND STUDENTS THAT < > IS A SINGLE LEXICAL ELEMENT.

3. THERE IS NO SUCH THING AS A GENERIC FORMAL RECORD TYPE.

THIS IS ANOTHER POSSIBLE BREAK POINT.

GENERIC FORMAL TYPES

GENERIC FORMAL INTEGER TYPES

```
type identifier is range <>;
```

GENERIC FORMAL FLOATING-POINT TYPES

```
type identifier is digits <> ;
```

GENERIC FORMAL FIXED-POINT TYPES

```
type identifier is delta <> ;
```

GENERIC FORMAL DISCRETE TYPES

```
type identifier is  ( < > );
```

GENERIC FORMAL ARRAY TYPES

```
type identifier is
   array ( index subtype { , index subtype } ) of component subtype ;
```

GENERIC FORMAL ACCESS TYPES

```
type identifier is access designated subtype ;
```

GENERIC FORMAL PRIVATE TYPES:

```
type identifier [ discriminant part ]  is [limited] private;
```

INSTRUCTOR NOTES

A GENERIC FORMAL PARAMETER DEFINES TWO VIEWS OF THE CLASS OF TYPES. ONE VIEW IS THE OPERATIONS ALLOWED WITHIN THE GENERIC UNIT. THE SECOND VIEW IS THE TYPES THAT MAY BE PASSED AS ACTUALS.

EXAMPLES OF THE LAST POINT WILL FOLLOW LATER. IF THE CLASS NEEDS EXAMPLES NOW, EXPLAIN THAT AN INTEGER SUBTYPE CAN BE PASSED TO A GENERIC FORMAL DISCRETE TYPE, SINCE ALL OPERATIONS THAT MAY BE APPLIED WITHIN THE GENERIC UNIT (INCLUDING 'Succ AND 'Pred ATTRIBUTES, FOR EXAMPLE) ARE AMONG THE OPERATIONS OF INTEGER TYPES.

MEANING OF A GENERIC FORMAL TYPE DECLARATION

```
type identifier is description of some class of types ;
```

- EVERY CLASS OF TYPES HAS A SET OF OPERATIONS ASSOCIATED WITH IT.

- WITHIN THE GENERIC UNIT, THE DECLARED IDENTIFIER STANDS FOR A TYPE HAVING ONLY THE OPERATIONS ASSOCIATED WITH THE DESCRIBED CLASS.

- IN A GENERIC INSTANTIATION, ONLY TYPES OR SUBTYPES WHOSE OPERATIONS INCLUDE THE OPERATIONS OF THE DESCRIBED CLASS MAY BE USED AS GENERIC ACTUAL PARAMETERS.

CONSEQUENCES:

- A GENERIC FORMAL TYPE CAN ONLY BE USED INSIDE THE GENERIC UNIT AS A TYPE OF THE DESCRIBED CLASS.

- IN AN INSTANTIATION, THE CORRESPONDING GENERIC ACTUAL TYPE CAN SOMETIMES BELONG TO A CLASS OTHER THAN THE DESCRIBED CLASS, AS LONG AS THAT CLASS HAS ALL THE NECESSARY OPERATIONS.

INSTRUCTOR NOTES

THE GENERIC FORMAL INTEGER TYPE DECLARATION LOOKS LIKE AN INTEGER TYPE DECLARATION, BUT WITH A BOX STANDING FOR AN UNSPECIFIED RANGE.

THE GENERIC FORMAL FLOATING-POINT TYPE DECLARATION LOOKS LIKE A FLOATING-POINT TYPE DECLARATION, BUT WITH A BOX STANDING FOR AN UNSPECIFIED NUMBER OF SIGNIFICANT DIGITS. (THE RANGE THAT IS OPTIONAL IN A FLOATING-POINT TYPE DECLARATION IS OMITTED FROM A GENERIC FORMAL FLOATING-POINT TYPE DECLARATION.)

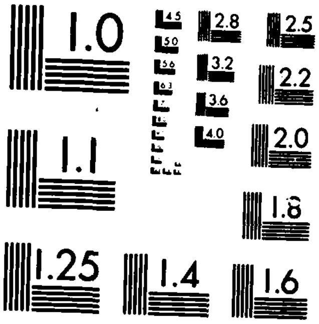
THE GENERIC FORMAL FIXED-POINT TYPE DECLARATION LOOKS LIKE A FIXED-POINT TYPE DECLARATION, BUT WITH A BOX STANDING FOR AN UNSPECIFIED DELTA. (THE RANGE THAT IS REQUIRED IN A FIXED-POINT TYPE DECLARATION IS OMITTED FROM A GENERIC FORMAL FIXED-POINT TYPE DECLARATION.)

THE USE OF GENERIC FORMAL FIXED-POINT TYPES IS ILLUSTRATED ON THE NEXT SLIDE. THE EXAMPLE IS REPRESENTATIVE FOR ALL THREE KINDS OF GENERIC FORMAL TYPES.

(LATER, AFTER GENERIC FORMAL PRIVATE TYPES ARE INTRODUCED, YOU WILL EXPLAIN HOW TO DECLARE A SINGLE GENERIC FORMAL TYPE THAT CAN BE MATCHED BY ANY NUMERIC TYPE.)

AD-A165 076 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA 3/7
TOPICS L305 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K506

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

GENERIC FORMAL PARAMETERS FOR NUMERIC TYPES

- GENERIC FORMAL INTEGER TYPES:

  type identifier is range <>;

  - CAN BE USED INSIDE THE GENERIC UNIT AS AN INTEGER TYPE
  - GENERIC ACTUAL PARAMETER CAN BE ANY INTEGER SUBTYPE

- GENERIC FORMAL FLOATING-POINT TYPES:

  type identifier is digits <>;

  - CAN BE USED INSIDE THE GENERIC UNIT AS A FLOATING-POINT TYPE
  - GENERIC ACTUAL PARAMETER CAN BE ANY FLOATING-POINT SUBTYPE

- GENERIC FORMAL FIXED-POINT TYPES:

  type identifier is delta <>;

  CAN BE USED INSIDE THE GENERIC UNIT AS A FIXED-POINT TYPE
  GENERIC ACTUAL PARAMETER CAN BE ANY FIXED-POINT TYPE

INSTRUCTOR NOTES

"*" IS NOT PREDEFINED FOR FIXED-POINT TYPES BECAUSE IT IS NOT CLEAR WHAT THE RESULT TYPE OF EACH MULTIPLICATION SHOULD BE.

THE CHOICE WE HAVE MADE HERE IS TO LET THE PRODUCT OF TWO VALUES IN THE SAME FIXED-POINT TYPE BELONG TO THE TYPE OF THE OPERANDS. HOWEVER, THIS CHOICE IS NOT IDEAL BECAUSE EACH MULTIPLICATION LOSES ACCURACY.

IT IS BECAUSE Fixed_Type IS A GENERIC FORMAL FIXED POINT TYPE THAT THE REAL LITERAL 1.0 CAN BE USED AS AN INITIAL VALUE FOR Result, THAT Result AND Left CAN BE MULTIPLIED (BUT ONLY INSIDE A TYPE CONVERSION), AND THAT THE CONVERSION TO Fixed_Type CAN BE WRITTEN.

THE THREE INSTANTIATIONS ARE LEGAL BECAUSE Temperature_Type, Distance_Type, and Duration ARE ALL FIXED-POINT TYPES.

GENERIC FORMAL PARAMETER FOR *FIXED-POINT TYPES* -- *EXAMPLE*

GENERIC UNIT:

```
generic
   type Fixed_Type is delta <>;
function Fixed_Power (Left : Fixed_Type; Right : Natural) return Fixed_Type;

function Fixed_Power (Left : Fixed_Type; Right : Natural) return Fixed_Type is
   Result : Fixed_Type := 1.0;
begin -- Fixed_Power
   for I in 1 .. Right loop
      Result := Fixed_Type (Result * Left);
   end loop;
   return Result;
end Fixed_Power;
```

CONTEXT FOR INSTANTIATIONS:

```
type Temperature_Type is delta 0.1 range 0.0 .. 373.0;
type Distance_Type is delta 0.005 range -1000.0 .. 1000.0;
```

INSTANTIATIONS OVERLOADING "**":

```
function "**" is new Fixed_Power (Fixed_Type =>  Temperature_Type);
function "**" is new Fixed_Power (Fixed_Type =>  Distance_Type);
function "**" is new Fixed_Power (Fixed_Type =>  Duration);
```

INSTRUCTOR NOTES

THE GENERIC FORMAL PARAMETER DECLARATION LOOKS LIKE AN ENUMERATION TYPE DECLARATION, BUT WITH A BOX STANDING FOR AN UNSPECIFIED LIST OF ENUMERATION LITERALS.

BULLET 2, SUBBULLET 2:

BE SURE THE CLASS UNDERSTANDS THE PRINCIPLE, WHICH IS VITAL IN UNDERSTANDING GENERIC TYPE PARAMETERS. THIS IS THE SIMPLEST AND CLEAREST EXAMPLE OF THE PRINCIPLE IN ALL OF Ada.

THE CONVERSE OF THIS SENTENCE IS NOT TRUE. THERE ARE OPERATIONS DEFINED FOR INTEGER TYPES BUT NOT FOR ENUMERATION TYPES (+, -, rem, mod, INTEGER LITERALS, NUMERIC TYPE CONVERSIONS, ETC.). THAT IS WHY ENUMERATION SUBTYPE CANNOT BE USED AS AN ACTUAL PARAMETER CORRESPONDING TO A GENERIC FORMAL INTEGER TYPE.

OTHER IMPORTANT USES OF DISCRETE TYPES, BESIDES THOSE LISTED ON THE SLIDE, INCLUDE THEIR USE AS ARRAY INDEX VALUES, For-Loop PARAMETERS, AND DISCRIMINANTS.

AN EXAMPLE OF THE USE OF GENERIC FORMAL DISCRETE TYPES FOLLOWS.

GENERIC FORMAL PARAMETER FOR DISCRETE TYPES

```
type identifier is (<>);
```

- CAN BE USED INSIDE THE GENERIC UNIT AS AN ENUMERATION TYPE
- GENERIC ACTUAL PARAMETER MAY BE
  - ANY ENUMERATION SUBTYPE
  - ANY INTEGER SUBTYPE, SINCE ALL OPERATIONS DEFINED FOR ENUMERATION TYPES ('Pos, 'Val, 'Image, 'Value, 'First, 'Last, 'Pred, 'Succ, >, <, ETC.) ARE ALSO DEFINED FOR INTEGER TYPES

INSTRUCTOR NOTES

THE FOLLOWING SLIDE ILLUSTRATES THE USE OF THIS GENERIC PACKAGE.

THE CORRESPONDING GENERIC BODY IS GIVEN ON THE SLIDE AFTER THAT.

GENERIC FORMAL PARAMETER FOR DISCRETE TYPES -- EXAMPLE

```
generic

   type Discrete_Type is ( <> );

package Cyclic_Operations_Package is

   function Cyclic_Successor (X : Discrete_Type) return Discrete_Type;
   function Cyclic_Predecessor (X : Discrete_Type) return Discrete_Type;

end Cyclic_Operations_Package;
```

MEANING OF FUNCTIONS:

- Cyclic_Successor IS LIKE Discrete_Type'Succ, EXCEPT THAT

  Cyclic_Successor (Discrete_Type'Last) = Discrete_Type'First

- Cyclic_Predecessor IS LIKE Discrete_Type'Pred, EXCEPT THAT

  Cyclic_Predecessor (Discrete_Type'First) = Discrete_Type'Last

INSTRUCTOR NOTES

THESE EXAMPLES SHOW Cyclic_Operations_Package INSTANTIATED WITH BOTH AN ENUMERATION TYPE AND AN INTEGER TYPE.

POINT OUT THAT THE use CLAUSES NAME THE INSTANCE, NOT THE TEMPLATE.

POSSIBLE INSTANTIATIONS OF Cyclic_Operations_Package

```
type Day_Type is
   (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

package Weekly_Cycle_Package is new
   Cyclic_Operations_Package (Discrete_Type =>  Day_Type);

use Weekly_Cycle_Package;
```

ASSUMING THE Day_Type VARIABLE Today HOLDS TODAY'S DAY:

- Cyclic_Successor (Today) IS TOMORROW'S DAY (EVEN IF Today = Saturday)

- Cyclic_Predecessor (Today) IS YESTERDAY'S DAY (EVEN IF Today = Sunday)

```
type Military_Hour_Type is range 0 .. 23;

package Military_Clock_Package is new
   Cyclic_Operations_Package (Discrete_Type =>  Military_Hour_Type);

use Military_Clock_Package;

Cyclic_Successor (12) = 13, Cyclic_Successor (23) = 0
Cyclic_Predecessor (13) = 12, Cyclic_Predecessor (0) = 23
Military_Hour_Type'(10) + 10 = 20
Military_Hour_Type'(20) + 10 = 6
Military_Hour_Type'(n) + 24 = n for any n in 0 .. 23
Military_Hour_Type'(8) - 50 = 6
Military_Hour_Type'(8) - 1 = 7
```

INSTRUCTOR NOTES

WALK THROUGH Cyclic_Successor, POINTING OUT THE USE OF DISCRETE ATTRIBUTES ('First, 'Last, 'Succ).

GIVE THE CLASS A FEW MOMENTS TO FILL IN THE BOXES IN Cyclic_Predecessor, THEN GO OVER THE ANSWERS:

```
function Cyclic_Predecessor (X : Discrete_Type) return Discrete_Type is
begin
   if X = Discrete_Type'First then
      return Discrete_Type'Last;
   else
      return Discrete_Type'Pred (X);
   end if;
end Cyclic_Predecessor;
```

GENERIC FORMAL PARAMETER FOR DISCRETE TYPES -- EXAMPLE (Continued)

```
package body Cyclic_Operations_Package is

   function Cyclic_Successor (X : Discrete_Type) return Discrete_Type is
   begin
      if X = Discrete_Type'Last then
         return Discrete_Type'First;
      else
         return Discrete_Type'Succ (X);
      end if;
   end Cyclic_Successor;

   function Cyclic_Predecessor (X : Discrete_Type) return Discrete_Type is
   begin

      if □ then

         return □;

      else

         return □;

      end if;
   end Cyclic_Predecessor;

end Cyclic_Operations_Package;
```

INSTRUCTOR NOTES

A GENERIC FORMAL UNCONSTRAINED ARRAY TYPE DECLARATION HAS PRECISELY THE SAME FORM AS AN ORDINARY UNCONSTRAINED ARRAY TYPE DECLARATION.

GENERIC FORMAL PARAMETER FOR UNCONSTRAINED ARRAY TYPES

```
type identifier is
   array ( type_mark range < > ,{ type_mark range <>})
      of type_mark [ constraint ];
   -  A type_mark is an identifier naming a type or subtype
```

- CAN BE USED INSIDE THE GENERIC UNIT AS AN UNCONSTRAINED ARRAY TYPE

- GENERIC ACTUAL PARAMETER MUST BE AN UNCONSTRAINED ARRAY TYPE WITH
  - SAME NUMBER OF DIMENSIONS
  - SAME INDEX TYPES IN CORRESPONDING DIMENSIONS
  - SAME COMPONENT SUBTYPE

INSTRUCTOR NOTES

GIVEN AN ARRAY (a_0, a_1, ..., a_n) AND A VALUE x, AN INSTANCE OF THIS GENERIC FUNCTION RETURNS THE VALUE $\sum_{i=0}^{n} a_0 x^{n-i}$: THE TYPE OF x AND THE ARRAY COMPONENTS IS PASSED AS THE FIRST GENERIC PARAMETER. THE TYPE OF THE ARRAY ITSELF IS PASSED AS THE SECOND GENERIC PARAMETER AND USED IN A FUNCTION PARAMETER SPECIFICATION. BECAUSE THE ARRAY TYPE IS UNCONSTRAINED, AN INSTANCE OF THIS GENERIC FUNCTION CAN HANDLE ARRAYS OF VARIOUS LENGTHS (CORRESPONDING TO POLYNOMIALS OF VARIOUS DEGREES).

THE NEXT SLIDE ADDRESSES THE FACT THAT THE SECOND GENERIC FORMAL PARAMETER IS DECLARED IN TERMS OF THE FIRST.

(THE EVALUATION OF THE POLYNOMIAL USES HORNER'S RULE. THE POLYNOMIAL

$$a_0x^3 + a_1x^2 + a_2x + a_3,$$

FOR EXAMPLE, IS EQUAL TO

$$(((0 * x + a_0) * x + a_2) * x + a_1) * x + a_0.$$

USING HORNER'S RULE, EVALUATION OF AN n-DEGREE POLYNOMIAL REQUIRES n + 1 MULTIPLICATIONS AND n ADDITIONS. THE DIRECT METHOD REQUIRES $n^2 + n$ MULTIPLICATIONS AND n ADDITIONS. YOU NEED NOT BRING THIS UP UNLESS SOMEONE QUESTIONS THE ALGORITHM.)

WE SHALL RETURN TO THIS EXAMPLE LATER AND GENERALIZE IT TO WORK FOR ANY NUMERIC TYPE.

GENERIC FORMAL PARAMETER FOR UNCONSTRAINED ARRAY SUBTYPES -- EXAMPLE 1

```
generic
   type Floating_Point_Type is digits <>;
   type Coefficient_List_Type is array (Positive range <> ) of Floating_Point_Type;
function Polynomial_Value
   (Coefficients : Coefficient_List_Type; X : Floating_Point_Type)
                    return Floating_Point_Type;

function Polynomial_Value
   (Coefficients : Coefficient_List_Type; X : Floating_Point_Type) return
    Floating_Point_Type is

   Sum : Floating_Point_Type := 0.0;

begin -- Polynomial_Value

   for I in Coefficients'Range loop
      Sum := X * Sum + Coefficients (I);
   end loop;

   return Sum;

end Polynomial_Value;
```

INSTRUCTOR NOTES

SUBPROGRAM PARAMETERS DO NOT WORK THIS WAY!

(THE RULES OF Ada STIPULATE THAT, UPON INSTANTIATION, GENERIC PARAMETERS ARE SUBSTITUTED IN THE ORDER THEY ARE WRITTEN. UPON A SUBPROGRAM CALL, SUBPROGRAM PARAMETERS MAY BE PROCESSED IN ANY ORDER. THUS ONE SUBPROGRAM FORMAL PARAMETER MAY NOT BE USED TO SPECIFY THE DEFAULT INITIAL VALUE OF A LATER PARAMETER.)

DECLARATION OF ONE GENERIC FORMAL PARAMETER IN TERMS OF ANOTHER

- A NAME DECLARED AS A GENERIC FORMAL PARAMETER MAY APPEAR IN A LATER GENERIC FORMAL PARAMETER DECLARATION.

- A GENERIC FORMAL PARAMETER IS REPLACED BY THE CORRESPONDING ACTUAL PARAMETER BEFORE SUBSEQUENT GENERIC PARAMETERS ARE PROCESSED..

- EXAMPLE:

```
generic
   type Floating_Point_Type is digits <>;
   type Coefficient_List_Type is
      array (Positive range <>) of Floating_Point_Type;
   function Polynomial_Value
      (Coefficients : Coefficient_List_Type; X : Floating_Point_Type)
       return Floating_Point_Type;

type Long_Float_List_Type is array (Positive range <>) of Long_Float;

function Long_Float_Polynomial_Value is
   new Polynomial_Value
      (Floating_Point_Type => Long_Float,
      Coefficient_List_Type => Long_Float_List_Type);
```

THE SECOND GENERIC FORMAL PARAMETER IS PROCESSED AS IF IT READ:

```
   type Coefficient_List_Type is array (Positive range <>) of  Long_Float ;
```

THUS IT MATCHES THE ACTUAL PARAMETER Long_Float_List_Type.

- THE ORDER OF THE GENERIC FORMAL PARAMETERS IS CRITICAL.

INSTRUCTOR NOTES

IN THIS EXAMPLE, BOTH THE INDEX TYPE AND THE COMPONENT TYPE OF THE GENERIC FORMAL ARRAY TYPE ARE GENERIC PARAMETERS DECLARED EARLIER.

TO INSTANTIATE Reverse_Order FOR A PARTICULAR UNCONSTRAINED ARRAY TYPE, THE INSTANTIATION MUST NAME THE INDEX AND COMPONENT TYPES, THEN THE ARRAY TYPE.

AN INDEX TYPE IN A GENERIC FORMAL ARRAY TYPE MUST BE EITHER AN ORDINARY DISCRETE SUBTYPE (LIKE Positive IN THE PREVIOUS EXAMPLE), A GENERIC FORMAL DISCRETE TYPE -- "is (< >)" -- OR A GENERIC FORMAL INTEGER TYPE.

THE DECLARATION OF Result NECESSARILY CONTAINS AN INDEX CONSTRAINT, SINCE Array_Type IS UNCONSTRAINED.

NOTE THE USE OF ATTRIBUTES OF THE FORMAL TYPE Index_Type. (USE OF 'Succ OR 'Pred IN THE LOOP WOULD HAVE RAISED THE POSSIBILITY OF Constraint_Error ON THE LAST PASS THROUGH THE LOOP, ARISING FROM AN ATTEMPT TO EVALUATE Index_Type'Succ (Index_Type'Last) OR Index Type'Pred (Index_Type'First).)

GENERIC FORMAL PARAMETER FOR UNCONSTRAINED ARRAY SUBTYPES -- EXAMPLE 2

```
generic
   type Index_Type is ( <> );
   type Component_Type is range <>;
   type Array_Type is array (Index_Type range <>) of Component_Type;
function Reverse_Order (Original_Array : Array_Type) return Array_Type;

function Reverse_Order (Original_Array : Array_Type) return Array_Type is

   Result                       : Array_Type (Original_Array'Range);
   Source_Index, Target_Index : Index_Type;
   First_Pos                    : constant := Index_Type'Pos (Original_Array'First);
   Last_Pos                     : constant := Index_Type'Pos (Original_Array'Last);

begin -- Reverse_Order

   for Offset in 0 .. Original_Array'Length - 1 loop
      Source_Index := Index_Type'Val (First_Pos + Offset);
      Target_Index := Index_Type'Val (Last_Pos - Offset);
      Result (Target_Index) := Original_Array (Source_Index);
   end loop;
   return Result;

end Reverse_Order;
```

INSTRUCTOR NOTES

THE GENERIC FORMAL PARAMETER FOR CONSTRAINED ARRAY SUBTYPES LOOKS LIKE AN ORDINARY CONSTRAINED ARRAY TYPE DECLARATION. HOWEVER, THE ONLY WAY AN INDEX TYPE MAY BE DESCRIBED IS BY A TYPE MARK, NOT BY A RANGE OR A TYPE MARK FOLLOWED BY A RANGE.

THE 'First, 'Last, 'Length, AND 'Range ATTRIBUTES MAY BE APPLIED TO THE FORMAL TYPE ITSELF OR TO OBJECTS IN THE TYPE.

GENERIC FORMAL PARAMETER FOR CONSTRAINED ARRAY SUBTYPES

```
type identifier is array ( type mark {, type mark }) of type mark [ constraint ];
```

- CAN BE USED INSIDE THE GENERIC UNIT AS A CONSTRAINED ARRAY TYPE.
- GENERIC ACTUAL PARAMETER MUST BE A CONSTRAINED ARRAY TYPE WITH
  - SAME NUMBER OF DIMENSIONS
  - SAME INDEX SUBTYPES IN CORRESPONDING DIMENSIONS
  - SAME COMPONENT SUBTYPE

INSTRUCTOR NOTES

Square_Matrix_Type CAN BE INSTANTIATED WITH ANY TWO-DIMENSIONAL CONSTRAINED ARRAY SUBTYPE WITH FLOATING-POINT COMPONENTS AND THE SAME INDICES IN BOTH DIMENSIONS.

SINCE Square_Matrix_Type IS CONSTRAINED, NO INDEX CONSTRAINT APPEARS ON THE DECLARATION OF RESULT.

GENERIC FORMAL PARAMETER FOR CONSTRAINED ARRAY SUBTYPES -- EXAMPLE

```
generic
   type Index_Subtype is (<>);
   type Component_Type is digits <>;
   type Square_Matrix_Type is
      array (Index_Subtype, Index_Subtype) of Component_Type;
function Transpose (Square_Matrix : Square_Matrix_Type) return Square_Matrix_Type;

function Transpose (Square_Matrix : Square_Matrix_Type) return Square_Matrix_Type is

   Result : Square_Matrix_Type;

begin -- Transpose

   for Row in Index_Subtype loop
      for Column in Index_Subtype loop
         Result (Row, Column) := Square_Matrix (Column, Row);
      end loop;
   end loop;
   return Result;

end Transpose;
```

INSTRUCTOR NOTES

A GENERIC FORMAL ACCESS TYPE DECLARATION LOOKS LIKE AN ORDINARY ACCESS TYPE DECLARATION, BUT THE TYPE MARK NAMING THE DESIGNATED SUBTYPE MAY NOT BE FOLLOWED BY A CONSTRAINT.

THE DESIGNATED SUBTYPE MAY BE ONE VISIBLE AT THE PLACE OF THE GENERIC DECLARATION OR ONE PASSED IN AS AN EARLIER GENERIC FORMAL TYPE.

SELECTION OF COMPONENTS OF DESIGNATED OBJECTS (EITHER BY SELECTED COMPONENTS OR INDEXED COMPONENTS) AND ATTRIBUTES OF DESIGNATED OBJECTS, ARE ALLOWED ONLY IF THE CORRESPONDING OPERATIONS ARE ALLOWED FOR THE DESIGNATED SUBTYPE.

USES ARE RARE BECAUSE THERE ARE NOT MANY USEFUL FUNCTIONS APPLICABLE TO ARBITRARY ACCESS TYPES.

ONE USE OF GENERIC FORMAL ACCESS TYPES IS IN THE PREDEFINED GENERIC PROCEDURE Unchecked Deallocation, WHICH WILL BE COVERED IN SECTION 13.

GENERIC FORMAL PARAMETER FOR ACCESS TYPES

type $\boxed{\texttt{identifier}}$ is access $\boxed{\texttt{type mark}}$;

- CAN BE USED INSIDE THE GENERIC UNIT AS AN ACCESS TYPE.

- GENERIC ACTUAL PARAMETER MUST BE AN ACCESS SUBTYPE POINTING TO VARIABLES IN THE SUBTYPE NAMED BY THE $\boxed{\texttt{type mark}}$.

- USES ARE RARE.

INSTRUCTOR NOTES

UNTIL NOW THE CONSEQUENCES OF THESE RULES HAVE BEEN UNTUITIVELY NATURAL.

WE REVIEW THESE RULES AS A PRELUDE TO GENERIC FORMAL PRIVATE AND LIMITED PRIVATE TYPES, FOR WHICH THE CONSEQUENCES OF THE RULES MAY BE SURPRISING.

REVIEW

- GENERIC FORMAL PARAMETER FOR A TYPE:

  type $\boxed{\text{identifier}}$ is $\boxed{\text{description of some class of type}}$;

- CAN BE USED INSIDE THE GENERIC UNIT AS A TYPE OF THE DESCRIBED CLASS

- GENERIC ACTUAL PARAMETER MUST BE A SUBTYPE WHOSE OPERATIONS INCLUDE ALL THE OPERATIONS FOR THAT CLASS OF TYPES

  - THE GENERIC ACTUAL PARAMETERS MAY HAVE OTHER OPERATIONS AS WELL (EXAMPLE: USING AN INTEGER SUBTYPE AS AN ACTUAL PARAMETER FOR A GENERIC FORMAL DISCRETE TYPE)

- THESE RULES GUARANTEE THAT ANY OPERATIONS APPLIED TO GENERIC FORMAL TYPES INSIDE THE GENERIC UNIT ARE REALLY DEFINED FOR THE GENERIC ACTUAL PARAMETER

INSTRUCTOR NOTES

THOUGH THE GENERIC FORMAL PARAMETER DECLARATION IS IDENTICAL IN FORM TO A PRIVATE TYPE DECLARATION IN A PACKAGE'S VISIBLE PART, ITS MEANING IS QUITE DIFFERENT.

- BULLET 1:
  - ITEM 4: AN EXAMPLE WILL BE GIVEN ON THE SLIDE AFTER THE NEXT ONE

- BULLET 2:
  - USES 1 AND 4 ARE ALSO AVAILABLE FOR LIMITED TYPES, BUT USES 2 AND 3 ARE NOT.

- BULLET 3: THE RESTRICTION IS NEEDED BECAUSE
  - ALLOCATION OF A VARIABLE IN AN UNCONSTRAINED ARRAY TYPE REQUIRES AN INITIAL VALUE OR AN INDEX CONSTRAINT
  - DECLARATION OF A VARIABLE IN AN UNCONSTRAINED ARRAY TYPE REQUIRES AN INDEX CONSTRAINT
  - ALLOCATION OF A VARIABLE IN AN UNCONSTRAINED RECORD TYPE WITHOUT DISCRIMINANTS REQUIRES AN INITIAL VALUE OR A DISCRIMINANT CONSTRAINT
  - DECLARATION OF A VARIABLE IN AN UNCONSTRAINED RECORD TYPE WITHOUT DISCRIMINANTS REQUIRES A DISCRIMINANT CONSTRAINT

EXCEPT IN THE CASE OF THIS RESTRICTION, THE LEGALITY OF AN INSTANTIATION CAN BE DETERMINED BY COMPARING THE GENERIC ACTUAL PARAMETERS WITH THE GENERIC FORMAL PARAMETERS. HOWEVER, THIS RESTRICTION INVOLVES A COMPARISON BETWEEN GENERIC ACTUAL PARAMETERS AND TEXT INSIDE THE GENERIC UNIT.

GENERIC FORMAL PRIVATE TYPES

```
type identifier is private;
```

- CAN BE USED INSIDE THE GENERIC UNIT IN THE SAME WAY AS A PRIVATE TYPE DECLARED ELSEWHERE:
  - OBJECTS AND SUBPROGRAM PARAMETERS CAN BE DECLARED
  - VALUES CAN BE ASSIGNED TO VARIABLES
  - VALUES CAN BE TESTED FOR EQUALITY AND INEQUALITY
  - IF THE TYPE IS USED AS A SUBPROGRAM PARAMETER TYPE IN THE DECLARATION OF A LATER GENERIC FORMAL SUBPROGRAM, VALUES CAN BE USED AS SUBPROGRAM PARAMETERS

- GENERIC ACTUAL PARAMETER MAY BE ANY SUBTYPE EXCEPT A LIMITED SUBTYPE
  - ANY TYPE EXCEPT A LIMITED TYPE CAN BE USED IN THE FOUR WAYS LISTED ABOVE

- A TECHNICAL RESTRICTION APPLIES IF THE GENERIC UNIT CONTAINS EITHER
  - DECLARATIONS OF OBJECTS IN THE TYPE
  - ALLOCATORS WITHOUT INITIAL VALUES FOR VARIABLES IN THE TYPE

  IN THIS CASE, THE CORRESPONDING GENERIC ACTUAL PARAMETER MAY NOT BE
  - AN UNCONSTRAINED ARRAY SUBTYPE
  - AN UNCONSTRAINED RECORD TYPE WITHOUT DEFAULT DISCRIMINANT VALUES

INSTRUCTOR NOTES

REMIND STUDENTS THAT THIS WAS THE EXAMPLE PRESENTED AT THE BEGINNING OF THE SECTION ON GENERIC UNITS.

Parameter_Type VARIABLES ARE INITIALIZED AND Parameter_Type VALUES ARE ASSIGNED. HOWEVER, NO OTHER PROPERTIES OF THE TYPE ARE ASSUMED.

Swap_Template MAY BE INSTANTIATED WITH ANY NON-LIMITED TYPE.

GENERIC FORMAL PRIVATE TYPES -- EXAMPLE 1

```
generic
   type Parameter_Type is private;
procedure Swap_Template (A, B : in out Parameter_Type);

procedure Swap_Template (A, B : in out Parameter_Type) is
   Old_A : constant Parameter_Type := A;
begin -- Swap_Template
   A := B;
   B := Old_A;
end Swap_Template;
```

INSTRUCTOR NOTES

THE GENERIC FORMAL PRIVATE TYPE IS USED AS A PARAMETER TYPE IN THE DECLARATION OF THE GENERIC FORMAL FUNCTION Is_To_Be_Kept.

THUS THE OPERATIONS AVAILABLE FOR Component_Type INSIDE THE GENERIC FUNCTION ARE ASSIGNMENT, TESTS FOR EQUALITY, AND THE FUNCTION Is_To_Be_Kept.

GENERIC FORMAL PRIVATE TYPES - EXAMPLE 2

```
generic
   type Component_Type is private;
   with function Is_To_Be_Kept (Item : Component_Type) return Boolean;
   type List_Type is array (Positive range<>) of Component_Type;
function Compression (List : List_Type) return List_Type;

-- AN INSTANCE OF Compression RETURNS AN ARRAY CONTAINING ONLY THOSE
-- COMPONENTS OF List FOR WHICH Is_to_Be_Kept IS TRUE, IN THE SAME ORDER.

function Compression (List : List_Type) return List_Type is

   Result_Buffer : List_Type (1 .. List'Length);
   Number_Kept   : Integer range 0 .. List'Length := 0;

begin
   for I in List'Range loop
      if Is_To_Be_Kept (List (I)) then                       -- Component_Type VALUE
                                                             -- USED AS A PARAMETER

         Number_Kept := Number_Kept + 1;
         Result_Buffer (Number_Kept) := List (I);            -- ASSIGNMENT OF A
                                                             -- Component_Type VALUE.

      end if;

   end loop;

   return Result_Buffer (1 .. Number_Kept);

end Compression;
```

INSTRUCTOR NOTES

GENERIC FORMAL LIMITED PRIVATE TYPES

```
type identifier is limited private;
```

- CAN BE USED INSIDE THE GENERIC UNIT IN THE SAME WAY AS A LIMITED PRIVATE TYPE DECLARED ELSEWHERE:
  - OBJECTS AND SUBPROGRAM PARAMETERS CAN BE DECLARED
  - VALUES CAN BE USED AS ACTUAL PARAMETERS WHEN CALLING APPROPRIATE GENERIC FORMAL SUBPROGRAMS

- GENERIC ACTUAL PARAMETER MAY BE ANY SUBTYPE
  - ANY SUBTYPE CAN BE USED IN THE TWO WAYS LISTED ABOVE

- SAME RESTRICTION APPLIES AS FOR GENERIC FORMAL PRIVATE TYPES

INSTRUCTOR NOTES

AN INSTANCE OF Process_Each_Component TAKES AN ARRAY AS A PARAMETER AND CALLS Process_One_Component WITH EACH COMPONENT OF THE ARRAY.

SINCE THE ONLY OPERATION APPLIED TO Component_Type INSIDE THE GENERIC UNIT IS A CALL ON THE GENERIC FORMAL PROCEDURE Process_One_Component, Component_Type CAN BE DECLARED LIMITED PRIVATE. THUS THE CORRESPONDING ACTUAL PARAMETER MAY BE ANY SUBTYPE, LIMITED OR NOT.

IN THIS CASE, THE ACTUAL PARAMETER IS NOT LIMITED. IN FACT, THE ACTUAL PROCEDURE CORRESPONDING TO Process_One_Component, Increment, IS IMPLEMENTED USING AN ASSIGNMENT STATEMENT. THIS IS FINE, SINCE THE DECLARATION OF Component_Type AS LIMITED PRIVATE RESTRICTS ONLY THE OPERATIONS THAT MAY BE ASSUMED FOR Component_Type INSIDE THE GENERIC UNIT.

GENERIC FORMAL LIMITED PRIVATE TYPES - EXAMPLE

```
generic
   type Component_Type is limited private;
   with procedure Process_One_Component (Component : in out Component_Type);
   type List_Type is array (Positive range <>) of Component_Type;
procedure Process_Each_Component (List : in out List_Type);

procedure Process_Each_Component (List : in out List_Type) is
begin
   for I in List'Range loop
      Process_One_Component (List (I));  -- THE ONLY OPERATION ON Component_Type
   end loop;
end Process_Each_Component
```

- CONTEXT:

```
procedure Increment (N : in out Integer) is
begin
   N := N + 1;
end Increment;

type Integer_List_Type is array (Positive range <>) of Integer;

Count_List : Integer_List_Type (1 .. 5) := (2, 4, 6, 8, 10);
```

- INSTANTIATION:

```
procedure Increment_List is new
   Process_Each_Component
      (Component_Type        => Integer,
       Process_One_Component => Increment,
       List_Type             => Integer_List_Type);
```

- USE:

```
Increment_List (Count_List);  -- ADDS 1 TO EACH COMPONENT OF Count_List.
```

INSTRUCTOR NOTES

- OPERATIONS
  - FOR LIMITED PRIVATE : JUST SUBPROGRAM CALLS
  - FOR PRIVATE : ABOVE PLUS =, /=, :=
  - FOR DISCRETE : ABOVE PLUS <,>, USE IN FOR LOOPS, USE AS ARRAY INDICES, ETC.
  - FOR INTEGER : ABOVE PLUS ARITHMETIC OPERATIONS AND INTEGER LITERALS

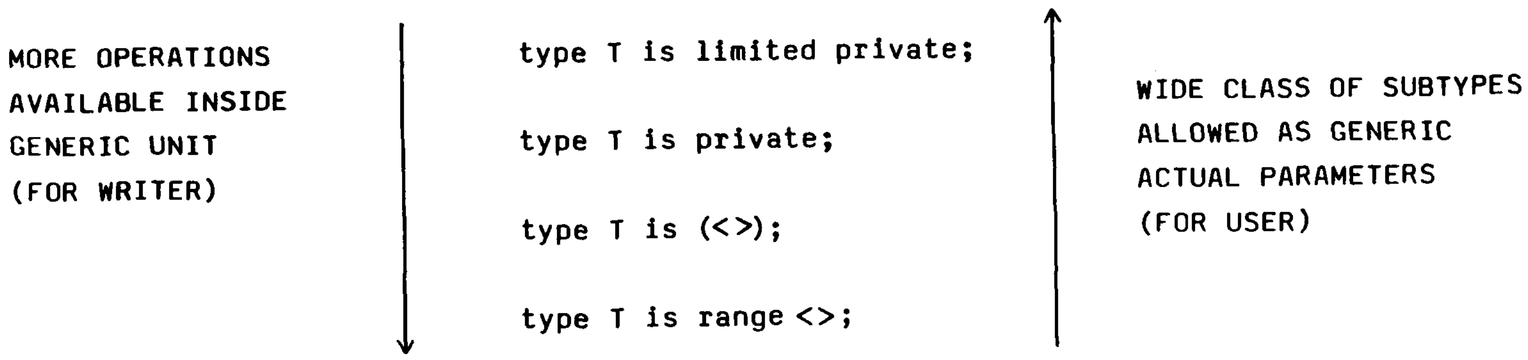
- SUBTYPES THAT CAN BE GENERIC ACTUAL PARAMETERS:
  - FOR INTEGER : INTEGER SUBTYPES
  - FOR DISCRETE : ABOVE PLUS ENUMERATION SUBTYPES
  - FOR PRIVATE : ABOVE PLUS OTHER NONLIMITED SUBTYPES
  - FOR LIMITED PRIVATE : ABOVE PLUS LIMITED SUBTYPES

A TRADEOFF

FLEXIBILITY IN <u>WRITING</u> A GENERIC UNIT

VERSUS

FLEXIBILITY IN <u>USING</u> A GENERIC UNIT



MORE OPERATIONS
AVAILABLE INSIDE
GENERIC UNIT
(FOR WRITER)

```
type T is limited private;

type T is private;

type T is (<>);

type T is range <>;
```

WIDE CLASS OF SUBTYPES
ALLOWED AS GENERIC
ACTUAL PARAMETERS
(FOR USER)

INSTRUCTOR NOTES

- THOUGH THERE ARE SOME COMPLICATED RULES, THIS IS A RELATIVELY MINOR FEATURE OF Ada. DO NOT DWELL ON IT.

- A DISCRIMINANT MUST BELONG TO A DISCRETE TYPE. IN THIS EXAMPLE, THE DISCRIMINANT BELONGS TO A GENERIC FORMAL DISCRETE TYPE DECLARED EARLIER.

- THE BAN ON DEFAULT INITIAL VALUES MEANS THAT OBJECTS IN THE GENERIC FORMAL TYPE MUST BE CONSTRAINED.

- THE USUAL RELATIONSHIP HOLDS BETWEEN OPERATIONS ALLOWED INSIDE THE GENERIC UNIT AND ALLOWABLE GENERIC ACTUAL PARAMETERS. A GENERIC FORMAL PRIVATE TYPE WITH DISCRIMINANTS CAN BE MATCHED BY ANY NONLIMITED TYPE WITH MATCHING DISCRIMINANTS. A GENERIC FORMAL LIMITED PRIVATE TYPE WITH DISCRIMINANTS CAN BE MATCHED BY ANY TYPE WITH MATCHING DISCRIMINANTS. (CONSTRAINED SUBTYPES OF A TYPE WITH DISCRIMINANTS MAY NOT BE GENERIC ACTUAL PARAMETERS IN EITHER CASE.)

- IN CONTRAST, AN ACTUAL PARAMETER MAY BE A TYPE WITH DISCRIMINANTS EVEN IF THE CORRESPONDING GENERIC FORMAL TYPE HAS NO DISCRIMINANTS. IN THIS CASE, THE ACTUAL PARAMETER HAS ALL THE OPERATIONS THAT ARE ALLOWED INSIDE THE GENERIC UNIT, PLUS OPERATIONS ON DISCRIMINANTS THAT ARE NOT ALLOWED. HOWEVER, THE TECHNICAL RESTRICTION NOTED EARLIER FOR GENERIC FORMAL PRIVATE TYPES THEN APPLIES: THE TYPE MAY NOT BE USED AS A GENERIC ACTUAL PARAMETER IF:
  - THE DECLARATION OF AN OBJECT IN THAT TYPE WOULD NORMALLY REQUIRE A DISCRIMINANT CONSTRAINT, AND
  - OBJECTS IN THE CORRESPONDING GENERIC FORMAL TYPE ARE CREATED INSIDE THE GENERIC UNIT

GENERIC FORMAL TYPES WITH DISCRIMINANTS

- GENERIC FORMAL PRIVATE AND LIMITED PRIVATE TYPES MAY HAVE DISCRIMINANTS

- THESE DISCRIMINANTS MAY NOT HAVE DEFAULT INITIAL VALUES

- EXAMPLE:

```
   generic
      type Discriminant_Type is (<>);
      type Allocated_Type (Discriminant : Discriminant_Type) is limited private;
      type Access_Type is access Allocated_Type;
      with procedure Set_Link  (Cell : in out Allocated_Type ; To : in Access_Type);
      with function Link_Value (Cell : Allocated_Type) return Access_Type;
   package Storage_Management_Package is
      procedure Allocate (Variant : in Discriminant_Type; Pointer : out Access_Type);
      procedure Free (Pointer : in out Access_Type);
   end Storage_Management_Package;
```

  (USES ARE RARE.)

- GENERIC ACTUAL PARAMETER MUST HAVE MATCHING DISCRIMINANTS.
  (SAME NUMBER AND SAME BASE TYPES)

- ADDITIONAL OPERATIONS AVAILABLE WITHIN THE GENERIC UNIT:
  - DISCRIMINANT CONSTRAINTS

```
        Pointer := new Allocated_Type (Discriminant => Variant);
```

  - EXAMINATION OF DISCRIMINANTS

```
        Set_Link (Pointer.all, Free_List (Pointer.Discriminant));
```

  - MEMBERSHIP TESTS

```
      subtype First_Subtype is Allocated_Type (Discriminant => Discriminant_Type'First);
      ...
      if Pointer in First_Subtype then
          ...
      end if;
```

INSTRUCTOR NOTES

THIS GENERIC FUNCTION WAS PRESENTED EARLIER AS THE FIRST EXAMPLE OF GENERIC FORMAL UNCONSTRAINED ARRAY TYPES.

THE SOLUTION IS GIVEN ON THE NEXT SLIDE. SEE IF CLASS MEMBERS CAN THINK OF THE SOLUTION WITHOUT LOOKING AHEAD.

GENERIC FORMAL PARAMETER FOR ARBITRARY NUMERIC TYPES

- HOW CAN WE GENERALIZE THE FOLLOWING WORK FOR ARBITRARY NUMERIC SUBTYPES RATHER THAN JUST FLOATING-POINT SUBTYPES?

```
generic
   type Floating_Point_Type is digits <>;
   type Coefficient_List_Type is array (Positive range < >) of Floating_Point_Type;
function Polynomial_Value
   (Coefficients : Coefficient_List_Type; X : Floating_Point_Type)
                  return Floating_Point_Type;

function Polynomial_Value
   (Coefficients : Coefficient_List_Type; X : Floating_Point_Type) return
    Floating_Point_Type is

   Sum : Floating_Point_Type := 0.0;

begin -- Polynomial_Value

   for I in Coefficients'Range loop
      Sum := X * Sum + Coefficients (I);
   end loop;

   return Sum;

end Polynomial_Value;
```

- THERE ARE MANY MATHEMATICAL FUNCTIONS LIKE THIS THAT CAN BE COMPUTED FOR ANY NUMERIC TYPE USING ONE ALGORITHM.

- EVEN THOUGH INTEGER, FLOATING-POINT, AND FIXED-POINT TYPES ALL HAVE OPERATIONS *, +, ETC., THERE IS NO GENERIC FORMAL PARAMETER SPECIFICALLY FOR NUMERIC TYPES.

INSTRUCTOR NOTES

THE GENERIC FORMAL FLOATING-POINT TYPE NAMED Floating_Point_Type HAS BEEN REPLACED BY A A GENERIC FORMAL PRIVATE TYPE NAMED Numeric_Type.

POINT OUT THE GENERIC PARAMETER ZERO. SINCE Numeric_Type IS A FORMAL PRIVATE TYPE, NUMERIC LITERALS FOR THE TYPE ARE NOT AVAILABLE. THUS THE VALUE TO BE USED TO INITIALIZE Sum MUST BE PASSED IN AS A GENERIC PARAMETER.

THE NAMES "+" AND "*" FOR THE FORMAL FUNCTIONS DO NOT MAKE THE FORMAL TYPE NUMERIC. THEY MAY BE MATCHED BY ANY ACTUAL FUNCTIONS WITH THE RIGHT PARAMETER AND RESULT TYPE. THE NAMES "+" AND "-" ARE CHOSEN TO INDICATE THAT WE THINK OF THE FIRST FORMAL FUNCTION AS ADDITION AND THE SECOND AS MULTIPLICATION.

INSTANTIATIONS ARE DEPICTED ON THE NEXT SLIDE.

SOLUTION : USE A GENERIC FORMAL PRIVATE TYPE

- TO WRITE A GENERIC UNIT APPLICABLE TO ANY NUMERIC TYPE, YOU MUST:
  - PASS IN THE TYPE AS A GENERIC FORMAL PRIVATE TYPE
  - PASS IN ANY REQUIRED ARITHMETIC OPERATIONS AS GENERIC FORMAL FUNCTIONS
  - PASS IN ANY REQUIRED NUMERIC VALUES AS GENERIC FORMAL CONSTANTS

```
generic
   type Numeric_Type is private;
   with function "+" (Left, Right : Numeric_Type) return Numeric_Type;
   with function "*" (Left, Right : Numeric_Type) return Numeric_Type;
   Zero : in Numeric_Type;
   type Coefficient_List_Type is array (Positive range <>) of Numeric_Type;
function Polynomial_Value
   (Coefficients : Coefficient_List_Type; X : Numeric_Type) return Numeric_Type;

function Polynomial_Value
   (Coefficients : Coefficient_List_Type; X : Numeric_Type) return Numeric_Type is

   Sum : Numeric_Type := Zero;

begin -- Polynomial_Value

   for I in Coefficients'Range loop
      Sum := X * Sum + Coefficients (I);  -- "*" AND "+" ARE GENERIC FORMAL FUNCTIONS
   end loop;

   return Sum;

end Polynomial_Value;
```

- NOTE THE USE OF OPERATOR SYMBOLS FOR GENERIC FORMAL FUNCTIONS.

INSTRUCTOR NOTES

Duration_Product IS NEEDED BECAUSE THE PREDEFINED VERSION OF "*" FOR MULTIPLYING TWO DURATION VALUES DOES NOT HAVE A RESULT OF THE SAME TYPE, AS REQUIRED BY THE DECLARATION OF THE GENERIC FORMAL FUNCTION "*". (TECHNICALLY, THE PRODUCT IS OF A SPECIAL TYPE NAMED universal_fixed, WHICH IS WHY IT MUST BE CONVERTED IMMEDIATELY TO AN ORDINARY NUMERIC TYPE.)

IN THE FIRST INSTANTIATION, WE SEE THAT THE FORMAL FUNCTION "+" (TO THE LEFT OF THE =>) IS MATCHED BY A VERSION OF "+" (TO THE RIGHT OF THE =>), WITH Integer OPERANDS AND AN Integer RESULT AND SIMILARLY FOR "*". IN THE SECOND INSTANTIATION, THE FORMAL FUNCTION "+" AND "*" MUST BE MATCHED BY VERSIONS WITH Float OPERANDS AND Float RESULTS, SO THE OPERATOR SYMBOLS TO THE RIGHT OF THE ARROW STAND FOR THE Float VERSIONS OF "+" AND "*". IN THE LAST EXAMPLE, THE FORMAL FUNCTION NAMED "*" IS MATCHED BY AN ACTUAL FUNCTION WITH A DIFFERENT NAME BUT THE REQUIRED PARAMETER AND RESULT TYPES.

LATER WE SHALL SEE A SHORTHAND ("is < > ") APPLICABLE WHEN FORMAL FUNCTIONS ARE EXPECTED TO MATCH ACTUAL FUNCTIONS WITH THE SAME NAME IN MOST INSTANTIATIONS. WHEN WE GET TO THAT POINT, WE SHALL REVISIT THIS EXAMPLE.

INSTANTIATIONS WITH VARIOUS NUMERIC TYPES

- CONTEXT:

```
   type Integer_List_Type is array (Positive range <>) of Integer;
   type Float_List_Type is array (Positive range <>) of Float;
   type Duration_List_Type is array (positive range <>) of Duration;

function Duration_Product (Left, Right : Duration) return Duration is
begin
   return Duration (Left * Right);      -- type conversion required for fixed-point
end Duration_Product;                   -- multiplication
```

- INSTANTIATIONS:

```
function Integer_Polynomial_Value is new
   Polynomial_Value
      (Numeric_Type          => Integer,
       "+"                   => "+",               -- Integer version of "+"
       "*"                   => "*",               -- Integer version of "+"
       Zero                  => 0,                 -- an integer literal
       Coefficient_List_Type => Integer_List_Type);

function Float_Polynomial_Value is new
   Polynomial_Value
      (Numeric_Type          => Float,
       "+"                   => "+",               -- Float version of "+"
       "*"                   => "*",               -- Float version of "+"
       Zero                  => 0.0,                  -- a real literal
       Coefficient_List_Type => Float_List_Type);

function Duration_Polynomial_Value is new
   Polynomial_Value
      (Numeric_Type          => Duration,
       "+"                   => "+",               -- Duration version of "+"
       "*"                   => Duration_Product,  -- no matching predefined version of "*"
       Zero                  => 0.0,               -- a real literal
       Coefficient_List_Type => Duration_List_Type);
```

INSTRUCTOR NOTES

- ANSWER 1:
  - CHANGE Numeric_Type TO A GENERIC FORMAL PRIVATE TYPE. ADD GENERIC FORMAL FUNCTIONS "+" AND "*".

    THE GENERIC DECLARATION WILL THEN READ AS FOLLOWS:

```
generic
   type Numeric_Type is private;
   with function "+" (Left, Right : Numeric_Type) return Numeric_Type;
   with function "*" (Left, Right : Numeric_Type) return Numeric_Type;
function Quadratic_Value (A, B, C, X : Numeric_Type) return Numeric_Type;
```

THERE IS NO NEED FOR A GENERIC FORMAL CONSTANT NAMED Zero. THOSE WHO DECLARED SUCH A CONSTANT COPIED THE SOLUTION FROM THE PREVIOUS SLIDE WITHOUT UNDERSTANDING.

- ANSWER 2:

```
function Type_Integer_Quadratic_Value is new
   Quadratic_Value (Numeric_Type =>  Integer, "+" => "+", "*" => "*");

function Type_Float_Quadratic_Value is new
   Quadratic_Value (Numeric_Type =>  Float, "+" => "+", "*" => "*");

function Duration_Quadratic_Value is new
   Quadratic_Value (Numeric_Type => Duration, "+" => "+", "*" => Duration_Product);
```

- ASK STUDENTS WHAT WOULD HAVE BEEN DIFFERENT IF THE return STATEMENT HAD READ

```
          return A*(X**2) + ...;
     INSTEAD OF
          return A*X*X+ ...;
```

ANSWER: IT WOULD HAVE BEEN NECESSARY TO PASS IN "**" AS A GENERIC PARAMETER. SINCE EXPONENTIATION IS NOT DEFINED FOR FIXED-POINT TYPES, IT WOULD HAVE BEEN NECESSARY TO DEFINE A FUNCTION, SAY Duration_Power ANALOGOUS TO Duration_Product.

EXERCISE

AN INSTANCE OF THE FOLLOWING GENERIC FUNCTION TAKES PARAMETERS A, B, C, AND X AND RETURNS THE VALUE $AX^2 + BX + C$:

```
generic
   type Numeric_Type is digits<>;

function Quadratic_Value (A, B, C, X : Numeric_Type) return Numeric_Type;

function Quadratic_Value (A, B, C, X : Numeric_Type) return Numeric_Type is
begin
   return A*X*X + B*X + C;
end Quadratic_Value;
```

1. GENERALIZE Quadratic_Value SO THAT IT WILL WORK FOR ANY NUMERIC TYPE.

2. SHOW HOW TO INSTANTIATE IT WITH THE FOLLOWING TYPES

   - Integer

   - Float

   - Duration (YOU MAY USE Duration_Product.)

INSTRUCTOR NOTES

ALL THREE INSTANTIATIONS ARE EQUIVALENT.

UNTIL NOW, WE HAVE SHOWN ONLY NAMED INSTANTIATIONS.

SYNTAX OF GENERIC INSTANTIATIONS

GENERIC INSTANTIATIONS MAY BE

- NAMED

```
function Type_Integer_Quadratic_Value is new
   Quadratic_Value (Numeric_Type =>  Integer, "+" => "+", "*" => "*");
```

- POSITIONAL

```
function Type_Integer_Quadratic_Value is new Quadratic_Value (Integer, "+", "*");
```

- MIXED

```
function Type_Integer_Quadratic_Value is new
   Quadratic_Value (Integer, "+" => "+", "*" => "*");
```

INSTRUCTOR NOTES

THESE ARE THE SAME RULES AS FOR SUBPROGRAM ACTUAL PARAMETERS. THEY ALSO RESEMBLE THE RULES FOR RECORD AGGREGATES.

NAMED, POSITIONAL, AND MIXED GENERIC INSTANTIATIONS

NAMED:

GENERIC PARAMETER ASSOCIATIONS OF THE FORM

$\boxed{\texttt{generic formal parameter}} \Rightarrow \boxed{\texttt{generic actual parameter}}$

MAY OCCUR IN ANY ORDER.

POSITIONAL:

GENERIC ACTUAL PARAMETERS ARE LISTED IN ORDER AND MATCHED WITH FORMAL PARAMETERS BASED ON POSITION.

MIXED:

POSITIONAL PARAMETERS COME FIRST, IN SEQUENCE, FOLLOWED BY NAMED PARAMETERS IN ANY ORDER.

INSTRUCTOR NOTES

AGAIN, THESE ARE THE SAME RULES AS FOR SUBPROGRAM PARAMETERS WITH DEFAULTS.

SPECIFICATION OF DEFAULT MEANINGS IS DESCRIBED IN THE FOLLOWING SLIDES.

DEFAULTS FOR GENERIC PARAMETERS

- THERE ARE WAYS TO SPECIFY DEFAULT MEANINGS FOR CERTAIN KINDS OF GENERIC FORMAL PARAMETERS.

- IF THERE IS A DEFAULT MEANING FOR A GIVEN GENERIC FORMAL PARAMETER, A CORRESPONDING GENERIC ACTUAL PARAMETER NEED NOT BE SUPPORTED BY AN INSTANTIATION.

- WHEN AN INSTANTIATION DOES NOT SPECIFY A MEANING FOR SOME GENERIC FORMAL PARAMETER, THE DEFAULT MEANING IS USED. WHEN AN INSTANTIATION DOES SPECIFY A MEANING, THAT MEANING OVERRIDES THE DEFAULT MEANING.

- WHEN A GENERIC ACTUAL PARAMETER IS OMITTED IN AN INSTANTIATION, ALL GENERIC ACTUAL PARAMETERS IN LATER POSITIONS MUST BE GIVEN IN NAMED FORM.

INSTRUCTOR NOTES

WE HAVE CIRCLED THE PARTS OF THE GENERIC FORMAL PARAMETER DECLARATION INTRODUCED FOR THE FIRST TIME ON THIS SLIDE.

WHEN SEVERAL GENERIC FORMAL CONSTANTS ARE DECLARED TOGETHER, THE DEFAULT EXPRESSION IS EVALUATED ONCE FOR EACH CORRESPONDING ACTUAL PARAMETER OMITTED FROM THE INSTANTIATION.

ENUMERATION LITERALS ARE CONSIDERED TO BE FUNCTIONS WITH NO PARAMETERS. THE DEFAULT MEANING FOR A GENERIC FORMAL PROCEDURE MAY ALSO BE AN ENTRY.

<> DEFAULTS FOR SUBPROGRAMS ARE DISCUSSED IN A LATER SLIDE.

GENERIC FORMAL PARAMETERS WITH DEFAULT MEANINGS

- GENERIC FORMAL CONSTANTS:

  $\boxed{\text{identifier}}\ \{,\ \boxed{\text{identifier}}\}:\ \text{in}\ \boxed{\text{type or subtype name}}\ :=\ \boxed{\text{expression}}\ ;$

  THE $\boxed{\text{expression}}$ GIVES THE DEFAULT MEANING OF EACH CONSTANT.

- DEFAULT MEANINGS CANNOT BE SPECIFIED FOR GENERIC FORMAL VARIABLES.

- GENERIC FORMAL SUBPROGRAM:

  $\text{with}\ \boxed{\text{subprogram specification}}\ \text{is}\ \boxed{\text{name}}\ ;$

  THE $\boxed{\text{name}}$ SPECIFIES A SUBPROGRAM OR ENUMERATION LITERAL COMPATIBLE WITH THE SPECIFICATION.

  THE SUBPROGRAM MUST BE ONE VISIBLE AT THE POINT OF THE GENERIC DECLARATION.

- DEFAULT MEANINGS CANNOT BE SPECIFIED FOR GENERIC FORMAL TYPES.

INSTRUCTOR NOTES

THE NEED FOR AND USE OF THE FUNCTION Matching_Keys ARE ILLUSTRATED ON THE NEXT SLIDE.

(THE PACKAGE ASSUMES THAT Matching_Keys IMPLEMENTS AN EQUIVALENCE RELATION, AND WILL NOT BEHAVE SENSIBLY IF IT DOES NOT. THAT IS, THE FOLLOWING MUST BE TRUE FOR ALL K1, K2, AND K3 IN Key_Type:

- Matching_Keys (K1, K1) = True
- Matching_Keys (K1, K2) = Matching_Keys (K2, K1)
- if Matching_Keys (K1, K2) = True and
  Matching_Keys (K2, K3) = True, then
  Matching_Keys (K1, K3) = True

THESE THREE CONDITIONS ARE EQUIVALENT TO THE ASSUMPTION GIVEN ON THE SLIDE THAT THE KEYS ARE DIVIDED INTO CLASSES SUCH THAT Matching_Keys (K1, K2) IS TRUE IF AND ONLY K1 AND K2 BELONG TO THE SAME CLASS.)

THE BODY OF Lookup_Table_Package IS NOT GIVEN ON A SLIDE BECAUSE IT IS NOT OUR MAIN CONCERN HERE. HOWEVER, IT WILL BE DISCUSSED AGAIN IN SECTION 5, UNDER SEARCHING.

NOTE: THE SLIDE DESCRIBES THE BEHAVIOR OF THE PACKAGE ABSTRACTLY IN TERMS OF THE GENERIC PARAMETERS. NOTHING IN THIS ABSTRACT DESCRIPTION REQUIRES THE TABLE TO BE IMPLEMENTED WITH A PHYSICAL SIZE GIVEN BY Table_Size. IN FACT, THE IMPLEMENTATION ON THE SUPPLEMENTAL HANDOUT USES A PHYSICAL SIZE OF Table_Size + 1 TO IMPLEMENT A LOGICAL SIZE OF Table_Size.

GENERIC FORMAL PARAMETERS WITH DEFAULT MEANINGS -- EXAMPLE

```
  generic
     type Key_Type is private;
     type Data_Type is private;
     Null_Data : in Data_Type;
    Table_Size : in Integer := 100;
    with function Matching_Keys (Key_1, Key_2 : Key_Type) return Boolean is "=";
 package Lookup_Table_Package is
    procedure Update_Data (Key : in Key_Type; Data : in Data_Type);
    procedure Look_Up_Data (Key : in Key_Type; Data : out Data_Type);
    Table_Full_Error : exception;
 end Lookup_Table_Package;
```

THIS PACKAGE PROVIDES OPERATIONS FOR STORING AND FETCHING DATA IDENTIFIED BY KEYS.

- KEYS ARE ASSUMED DIVIDED INTO CLASSES SUCH THAT Matching_Keys (K1, K2) IS TRUE IF AND ONLY IF K1 AND K2 ARE IN THE SAME CLASS. BY DEFAULT, Matching_Keys (K1, K2) MEANS K1 = K2, MEANING EACH Key_Type VALUE IS IN A CLASS BY ITSELF. AT MOST ONE DATA VALUE MAY BE ASSOCIATED WITH A GIVEN CLASS OF KEYS AT ONE TIME.

- INITIALLY, Null_Data IS IMPLICITLY ASSOCIATED WITH EACH CLASS OF KEYS.

- Update_Data EXPLICITLY ASSOCIATES A NEW Data_Type VALUE WITH A PARTICULAR CLASS OF KEYS, POSSIBLY OVERWRITING A PREVIOUS ASSOCIATION.

- Look_Up_Data FETCHES THE CURRENT DATA ASSOCIATED WITH A PARTICULAR CLASS OF KEYS.

- THERE MAY BE EXPLICIT ASSOCIATIONS FOR UP TO Table_Size DIFFERENT CLASSES OF KEYS. ONCE THIS BOUND IS REACHED, A CALL ON Update_Data WITH A KEY IN A NEW CLASS RAISES Table_Full_Error.

INSTRUCTOR NOTES

THIS IS A SIMPLIFIED EXAMPLE. IN PRACTICE, A FILE DIRECTORY CONTAINS MORE INFORMATION THAN JUST THE SIZE OF EACH FILE. THE ACTUAL TYPE CORRESPONDING TO Data_Type IS MORE LIKELY TO BE A RECORD TYPE INCLUDING COMPONENTS FOR TIME OF LAST UPDATE, ACCESS PRIVILEGES, AND SO FORTH.

SEPARATE INSTANTIATIONS OF GENERIC TEMPLATES PRODUCE SEPARATE PACKAGES WITH THEIR OWN COPIES OF ANY LOCAL VARIABLES, EVEN IF THE TWO INSTANTIATIONS HAVE IDENTICAL GENERIC ACTUAL PARAMETERS. A CALL ON File_Directory_Package.Look_Up_Data CANNOT AFFECT THE RESULT OF A CALL ON File_Directory_Package.Update_Data, FOR INSTANCE. (IT IS POSSIBLE TO WRITE A GENERIC PACKAGE WITH NO PARAMETERS AND INSTANTIATE IT TWICE TO PRODUCE TWIN PACKAGES WITH INDIVIDUAL COPIES OF LOCAL DATA.)

IT WOULD HAVE BEEN POSSIBLE TO OMIT THE Table_Size PARAMETER IN THE THIRD INSTANTIATION (LETTING IT DEFAULT TO 100) AND STILL SPECIFIED THE Matching_Keys PARAMETER. THE Matching_Keys PARAMETER COULD NEVER BE GIVEN IN POSITIONAL FORM IN SUCH A CASE.

SINCE Matching_Keys DEFAULTS TO "=", PHYSICALLY DISTINCT KEYS ARE CONSIDERED LOGICALLY DISTINCT BY DEFAULT (I.E., WE MAY DISREGARD THE NOTION OF CLASSES OF KEYS).

GENERIC FORMAL PARAMETERS WITH DEFAULT MEANINGS -- EXAMPLE

```
subtype File_Name_Subtype is String (1 .. 11);
subtype File_Size_Subtype is Integer range - 1 .. Integer'Last,
   -- A FILE SIZE OF -1 MEANS THE FILE DOES NOT EXIST.

package File_Directory_Packages is new
   Lookup_Table_Package
      (Key_Type =>  File_Name_Subtype,
       Data_Type =>  File_Size_Subtype,
       Null_Data =>  -1);

package Archived_File_List_Package is new
   Lookup_Table_Package
      (File_Name_Subtype, File_Size_Subtype, -1, 1000);
```

FILE NAMES
"ABC" AND "abc'
ARE CONSIDERED
DISTINCT

ASSUME WE HAVE A FUNCTION TO DETERMINE WHETHER TWO STRINGS ARE THE SAME WHEN UPPER AND LOWER CASE LETTERS ARE CONSIDERED EQUIVALENT.

```
Equivalent_Strings ("ADA", "Ada") = True
```

ASSUME WE HAVE A TYPE Line_Number_List_Type, CONSISTING OF LISTS OF INTEGERS, WITH A CONSTANT Empty_List.

```
subtype FORTRAN_Identifier_Type is String (1 .. 6);

package Variable_Cross_Reference_Package is new
   Lookup_Table_Package
      (Key_Type =>  FORTRAN_Identifier_Type,
       Data_Type =>  Line_Number_List_Type,
       Null_Data =>  Empty_List,
       Table_Size =>  2500,
       Matching_Keys =>  Equivalent_Strings);
```

FORTRAN IDENTIFIERS
"ABC" AND "abc"
ARE CONSIDERED
EQUIVALENT

INSTRUCTOR NOTES

THIS IS THE "SHORTHAND" ALLUDED TO EARLIER IN THE TEACHER'S GUIDE.

IN THE FORM OF DEFAULT CONSIDERED PREVIOUSLY, THE DEFAULT REFERRED TO A PARTICULAR SUBPROGRAM VISIBLE AT THE POINT OF THE GENERIC DECLARATION. THE "is < > " DEFAULT REFERS TO SUBPROGRAMS VISIBLE AT POINTS OF GENERIC INSTANTIATIONS. IDENTICAL INSTANTIATIONS AT DIFFERENT PLACES MAY RESULT IN DIFFERENT DEFAULT SUBPROGRAMS USING THIS FORM.

DEFAULT MEANINGS FOR GENERIC FORMAL SUBPROGRAMS
BASED ON NAME AND SUBPROGRAM SPECIFICATION

- **FORM:**

```
with subprogram specification is < >;
```

- **MEANING:**

IF THE CORRESPONDING GENERIC ACTUAL PARAMETER IS OMITTED, THERE MUST BE EXACTLY ONE SUBPROGRAM WITH THE SAME NAME, PARAMETER TYPES, AND (FOR FUNCTIONS) RESULT TYPE VISIBLE AT THE POINT OF GENERIC INSTANTIATION.

THIS SUBPROGRAM IS TAKEN AS THE MEANING OF THE GENERIC FORMAL PARAMETER.

INSTRUCTOR NOTES

THIS EXAMPLE IS BASED ON THE EXERCISE A FEW SLIDES EARLIER.

REMIND STUDENTS THAT Duration_Product IS A FUNCTION TAKING TWO DURATION PARAMETERS AND RETURNING A DURATION RESULT.

IN THE SECOND INSTANTIATION, THE GENERIC ACTUAL PARAMETER FOR "*" MUST BE NAMED BECAUSE THE PRECEDING GENERIC ACTUAL PARAMETER HAS BEEN OMITTED.

EXAMPLE OF A < > DEFAULT MEANING FOR A GENERIC FORMAL SUBPROGRAM

```
generic
   type Numeric_Type is private;
   with function "+" (Left, Right : Numeric_Type) return Numeric_Type is < > ;
   with function "*" (Left, Right : Numeric_Type) return Numeric_Type is < > ;
function Quadratic_Value (A, B, C, X : Numeric_Type) return Numeric_Type;
```

- THE INSTANTIATION

```
function Type_Integer_Quadratic_Value is new
   Quadratic_Value (Integer);
```

IS EQUIVALENT TO

```
function Type_Integer_Quadratic_Value is new
   Quadratic_Value
      (Numeric_Type =>  Integer,
       "+"          =>  "+",        -- "+" FOR TYPE Integer
       "*"          =>  "*");       -- "*" FOR TYPE Integer
```

- THE INSTANTIATION

```
function Duration_Quadratic_Value is new
   Quadratic_Value (Duration, "*" =>  Duration_Product);
```

IS EQUIVALENT TO:

```
function Duration_Quadratic_Value is new
   Quadratic_Value
      (Numeric_Type =>  Duration,
       "+"          =>  "+",        -- "+" FOR TYPE Duration
       "*"          =>  Duration_Product);
```

INSTRUCTOR NOTES

THE CHANGES HAVE BEEN CIRCLED. IT MAKES SENSE TO TALK ABOUT MINIMA AND MAXIMA PRECISELY FOR THOSE TYPES WHICH HAVE AN OPERATION NAMED, OR ANALOGOUS TO, "<". AS THE HINT ABOUT Varying_String_Type INDICATES, THIS MAY INCLUDE LIMITED TYPES. (THE PACKAGE BODY MAKES NO USE OF =, /=, OR :=.)

```
generic
   type Item_Type is limited private;
   with function "<" (Left, Right : Item_Type) return Boolean is <>;
package Min_Max_Package is
   function Minimum (Item_1, Item_2 : Item_Type) return Item_Type;
   function Maximum (Item_1, Item_2 : Item_Type) return Item_Type;
end Min_Max_Package;

package body Min_Max_Package is

   function Minimum (Item_1, Item_2 : Item_Type) return Item_Type is
   begin
      if Item_1 < Item_2 then
         return Item_1;
      else
         return Item_2;
      end if;
   end Minimum;

   function Maximum (Item_1, Item_2 : Item_Type) return Item_Type is
   begin
      if Item_2 < Item_1 then
         return Item_1;
      else
         return Item_2;
      end if;
   end Maximum;

end Min_Max_Package;
```

WRITTEN THIS WAY INSTEAD OF Item_1 > Item_2 TO AVOID THE NEED TO PASS ">" AS A SEPARATE GENERIC PARAMETER

EXERCISE

THIS GENERIC PACKAGE PROVIDES MINIMUM AND MAXIMUM FUNCTIONS FOR ANY DISCRETE TYPE. GENERALIZE IT TO WORK FOR ANY TYPE FOR WHICH THE CONCEPTS OF MINIMUM AND MAXIMUM MAKE SENSE.

HINT: IT DOES NOT, IN GENERAL, MAKE SENSE TO TALK ABOUT THE MINIMUM OF TWO ACCESS VALUES. IT DOES MAKE SENSE TO TALK ABOUT THE MINIMUM OF TWO Varying_String_Type VALUES.

```
generic
   type Item_Type is ( < > );          -- < and > AVAILABLE
package Min_Max_Package is
   function Minimum (Item_1, Item_2 : Item_Type) return Discrete_Type;
   function Maximum (Item_1, Item_2 : Item_Type) return Discrete_Type;
end Min_Max_Package;

package body Min_Max_Package is

   function Minimum (Item_1, Item_2 : Item_Type) return Discrete_Type is
   begin
      if Item_1 < Item_2 then
         return Item_1;
      else
         return Item_2;
      end if;
   end Minimum;

   function Maximum (Item_1, Item_2 : Item_Type) return Discrete_Type is
   begin
      if Item_1 > Item_2 then
         return Item_1;
      else
         return Item_2;
      end if;
   end Maximum;

end Min_Max_Package;
```

INSTRUCTOR NOTES

SECTION 12

DERIVED TYPES

INSTRUCTOR NOTES

SOME USES FOR DERIVED TYPES ARE DESCRIBED ON LATER SLIDES.

TYPE CONVERSION IS ALLOWED BETWEEN ANY TWO TYPES WITH COMMON ANCESTRY, E.G., BETWEEN A DERIVED TYPE AND ITS PARENT, BETWEEN TWO TYPES DERIVED FROM THE SAME TYPE OR BETWEEN A TYPE DERIVED FROM A DERIVED TYPE AND ITS "GRANDPARENT TYPE."

DERIVED TYPES

- A DERIVED TYPE IS ESSENTIALLY A COPY OF ANOTHER TYPE.

- THE OTHER TYPE IS CALLED THE PARENT TYPE.

- DERIVED TYPE DECLARATION:

$$\texttt{type}\ \underbrace{\boxed{\texttt{identifier}}}_{\text{the derived type}}\ \texttt{is new}\ \underbrace{\boxed{\texttt{type or subtype name}}\ \left[\boxed{\texttt{constraint}}\right]}_{\text{a subtype of the parent type}};$$

- VALUES OF A DERIVED TYPE:

  FOR EACH VALUE IN THE PARENT TYPE, THERE IS AN IDENTICAL VALUE IN THE DERIVED TYPE.

- OPERATIONS OF A DERIVED TYPE:

  -- ALL PREDEFINED OPERATIONS OF THE PARENT TYPE

  -- TYPE CONVERSION BETWEEN ANY TWO TYPES IN THE EQUIVALENCE CLASS OF TYPES DERIVED FROM A TYPE

  -- IF THE PARENT TYPE IS PROVIDED BY A PACKAGE, A NEW VERSION OF EACH SUBPROGRAM PROVIDED BY THE PACKAGE THAT HAS A RESULT OR AT LEAST ONE PARAMETER OF THE PARENT TYPE. THE NEW VERSION USES THE DERIVED TYPE IN PLACE OF THE PARENT TYPE.

INSTRUCTOR NOTES

GIVEN THE DECLARATIONS:

```
    type T2 is new T1;
    type T3 is new T2;
```

(WHICH MAY NOT OCCUR WITHIN THE VISIBLE PART OF ONE PACKAGE) ALL DERIVED SUBPROGRAMS OF T2 ALSO BECOME DERIVED SUBPROGRAMS OF T3.

THE SPECIFICATION OF EACH DERIVED SUBPROGRAM ON THE SLIDE IS OBTAINED BY REPLACING EACH OCCURRENCE OF THE NAME OF THE PARENT TYPE (Queue_Type) WITH THE NAME OF THE DERIVED TYPE (Event_Queue_Type) IN THE SUBPROGRAM SPECIFICATIONS OF THE PACKAGE'S VISIBLE PART.

THE IMPLICIT DECLARATIONS SHOWN ON THE SLIDE CAN BE OVERRIDDEN BY EXPLICIT DECLARATIONS OF NEW SUBPROGRAMS WITH THE SAME NAMES, PARAMETER TYPES, AND RESULT TYPES.

BECAUSE Enqueue, FOR EXAMPLE, IS IMPLICITLY DECLARED, JUST AFTER THE DERIVED TYPE DECLARATION, IT IS REFERRED TO WITHIN Simulate AS Enqueue RATHER THAN Integer_Queue_Package.Enqueue. THE LATTER NAME REFERS ONLY TO THE VERSION OF Enqueue TAKING A Queue_Type PARAMETER.

DERIVED SUBPROGRAMS

- WHEN THE PARENT TYPE IS PROVIDED BY A PACKAGE, THE NEW VERSIONS OF THE PACKAGE'S VISIBLE SUBPROGRAMS ARE CALLED DERIVED SUBPROGRAMS.

- A DERIVED SUBPROGRAM IS IMPLICITLY DECLARED JUST AFTER THE DERIVED TYPE DECLARATION. IT IS REFERRED TO WITHOUT NAMING THE PACKAGE.

```
              package Integer_Queue_Package is
                 type Queue_Type is private;
                 procedure Enqueue (Queue : in out Queue_Type; Item : in Integer);
                 procedure Dequeue (Queue : in out Queue_Type; Item : out Integer);
                 function Is_Empty (Queue : Queue_Type) return Boolean;
                 function New_Empty_Queue return Queue_Type;
                 ...
              private
                 ...
              end Integer_Queue_Package;

              with Integer_Queue_Package;

              procedure Simulate is
                 ...
                 type Event_Queue_Type is new Integer_Queue_Package.Queue_Type;
                 procedure Enqueue (Queue : in out Event_Queue_Type; Item : in Integer);
IMPLICIT         procedure Dequeue (Queue : in out Event_Queue_Type; Item : out Integer);
DECLARATIONS     function Is_Empty (Queue : Event_Queue_Type) return Boolean;
                 function New_Empty_Queue return Event_Queue_Type;
                 Event_Queue : Event_Queue_Type;
                 ...
              begin -- Simulate
                 ...
                 Enqueue (Event_Queue, 0);  -- NOT Integer_Queue_Package.Enqueue
                 ...
              end Simulate;
```

INSTRUCTOR NOTES

DERIVED VERSUS INDEPENDENTLY-DECLARED ACCESS TYPES

```
type T1 is access Integer;
type T2 is access Integer;
type T3 is new T2;
```

- VALUES IN T1 AND T2 POINT TO DISJOINT SETS OF ALLOCATED VARIABLES.

  (A T1 ACCESS VALUE AND A T2 ACCESS VALUE NEVER DESIGNATE THE SAME ALLOCATED VARIABLE.)

- VALUES IN T3 ARE COPIES OF THOSE IN T2 -- POINTERS TO THE ALLOCATED VARIABLES THAT CAN BE DESIGNATED BY ACCESS VALUES IN T2.

  TYPE CONVERSION FROM T2 TO T3 PRODUCES ANOTHER POINTER TO THE SAME ALLOCATED VARIABLE, BUT VIEWED AS A VALUE OF TYPE T3.

INSTRUCTOR NOTES

DERIVED TYPES ARE USEFUL IN A VARIETY OF PECULIAR SITUATIONS.

THE FOLLOWING SLIDES DESCRIBE EACH OF THE FOUR LISTED SITUATIONS IN MORE DETAIL.

SOME USES OF DERIVED TYPES

- MULTIPLE ABSTRACTIONS

  CONSTRUCTING A DATA TYPE WHOSE VALUES ARE EACH REPRESENTED BY A SINGLE VALUE IN SOME OTHER TYPE

- MULTIPLE VERSIONS OF OPERATORS

  PROVIDING A PRIVATE TYPE WITH NEW VERSIONS OF OPERATORS, BUT KEEPING THE ORIGINAL VERSIONS AVAILABLE FOR USE INSIDE THE PACKAGE

- PRIVATE TYPES IMPLEMENTED BY GENERIC INSTANTIATION

  IMPLEMENTING A PRIVATE TYPE BY INSTANTIATING A GENERIC PACKAGE INSIDE A PRIVATE PART AND USING ONE OF THE TYPES PROVIDED BY THE INSTANCE

- MULTIPLE REPRESENTATIONS

  ALLOWING MULTIPLE INTERNAL REPRESENTATIONS OF THE SAME ABSTRACT DATA

INSTRUCTOR NOTES

USES OF DERIVED TYPES -- MULTIPLE ABSTRACTIONS

- OCCASIONALLY, WE DEFINE A NEW TYPE WHOSE UNDERLYING REPRESENTATION CONSISTS OF A SINGLE VALUE IN SOME OTHER TYPE.
- FROM AN ABSTRACT POINT OF VIEW, THESE ARE TWO DISTINCT TYPES AND THEIR USES SHOULD NOT BE INTERMIXED.
- IF WE DECLARE THE NEW TYPE TO BE DERIVED FROM THE OLD TYPE, THE COMPILER WILL ENFORCE THIS DISTINCTION.
- EXAMPLE: A DEPARTMENT STORE MUST KEEP TRACK OF THE CURRENT FISCAL QUARTER AND THE SEASON FOR WHICH CLOTHES ARE NOW BEING ORDERED. THESE TWO KINDS OF VALUES CAN BE REPRESENTED IN TERMS OF THE SAME ENUMERATION TYPE.

```
package Season_Package is
   type Season_Type is (Winter, Spring, Summer, Fall);
   function Season_After (Season : Season_Type) return Season;
   ...
end Season_Package;

type Fashion_Season_Type is new Season_Package.Season_Type;
type Fiscal_Quarter_Type is new Season_Package.Season_Type;

-- USES OF Season_Package.Season_Type, Fashion_Season_Type, AND
-- Fiscal_Quarter_Type CANNOT BE INTERMIXED.

-- EACH TYPE HAS ITS OWN VERSION OF Season_After.
```

INSTRUCTOR NOTES

THE COMPLICATION REFERRED TO IN BULLET 2 IS EXPLAINED ON THE NEXT SLIDE.

USES OF DERIVED TYPES -- MULTIPLE VERSIONS OF OPERATORS

- THE PROBLEM:

  -- SUPPOSE WE ARE DEFINING A LIMITED PRIVATE TYPE List_Type FOR LISTS OF INTEGERS. LISTS ARE TO BE IMPLEMENTED AS LINKED LISTS.

  -- THE PREDEFINED "=" TELLS US WHETHER TWO List_Type VALUES ARE ACCESS VALUES DESIGNATING THE SAME LIST CELL, NOT WHETHER THEY REPRESENT LISTS CONTAINING THE SAME SEQUENCE OF INTEGERS.

  -- THEREFORE, WE MAKE List_Type <u>limited</u> private AND PROVIDE OUR OWN VERSION OF "=".

```
package List_Package is
   type List_Type is limited private;
   ...
   function "=" (Left, Right : List_Type) return Boolean;
   ...
private
   type List_Cell_Type is
      record
         Integer_Part : Integer;
         Link_Part    : List_Type;
      end record;
   type List_Type is access List_Cell_Type;
end List_Package;
```

- THERE IS A SERIOUS COMPLICATION WITH WRITING THE BODY OF THE NEW VERSION OF "=".

THIS FUNCTION SIMPLY WALKS DOWN BOTH LINKED LISTS SIMULTANEOUSLY, CHECKING WHETHER CORRESPONDING LIST CELLS CONTAIN THE SAME INTEGER. IF THE RIGHT LIST RUNS OUT FIRST OR A MISMATCH IS FOUND, THE return STATEMENT INSIDE THE LOOP RETURNS False. IF THE END OF THE LEFT LIST IS REACHED BEFORE THIS HAPPENS, WE LEAVE THE LOOP AND CHECK THE RIGHT LIST. THE LISTS ARE IDENTICAL IF AND ONLY IF WE HAVE REACHED THE END OF THE RIGHT LIST AT THE SAME TIME.

THE DECLARATION OF A FUNCTION "=" WITH PARAMETERS OF TYPE List_Type HIDES THE VERSION OF "=" IMPLICITLY DECLARED BY THE DECLARATION OF List_Type IN THE PRIVATE PART. UNFORTUNATELY, IT IS THE HIDDEN VERSION OF "=" THAT WE WANT TO INVOKE TO COMPARE List_Type VALUES AS ACCESS VALUES RATHER THAN AS ABSTRACT SEQUENCES OF INTEGERS.

THE OPERATOR /= IN THE WHILE CONDITION ALSO LEADS TO A RECURSIVE CALL, BECAUSE DEFINING A NEW VERSION OF "=" IMPLICITLY DEFINES A NEW VERSION OF "/=" THAT CALLS "=" AND RETURNS THE OPPOSITE RESULT. RECURSION CONTINUES INDEFINITELY WITH THE FIRST PARAMETER EQUAL TO THE ORIGINAL VALUE OF Left AND THE SECOND PARAMETER EQUAL TO null.

USES OF DERIVED TYPES -- MULTIPLE VERSIONS OF OPERATORS (Continued)

```
function "=" (Left, Right : List_Type) return Boolean is

   Left_Cell_Pointer : List_Type := Left_Cell;
   Right_Cell_Pointer : List_Type := Right_Cell;

begin -- "="

   while Left_Cell_Pointer /= null loop

      if Right_Cell_Pointer = null or else
         Left_Cell_Pointer.Integer_Part /= Right_Cell_Pointer.Integer_Part then
         return False;
      else
         Left_Cell_Pointer := Left_Cell_Pointer.Link_Part;
         Right_Cell_Pointer := Right_Cell_Pointer.Link_Part;
      end if;

   end loop;

   return Right_Cell_Pointer = null;

end "=";
```

- THE STING! IN THE *if* STATEMENT AND THE BOTTOM *return* STATEMENT, WE WANT = TO COMPARE ACCESS VALUES, BUT THESE OCCURRENCES OF THE = OPERATOR ARE REALLY RECURSIVE CALLS ON THE NEW VERSION OF "=".

  THIS IS OBVIOUSLY NOT WHAT WE WANT. THE RECURSION NEVER STOPS.

List_Type IS DERIVED FROM List_Cell_Pointer_Type.

A NEW VERSION OF "=" IS DEFINED FOR List_Type, BUT THE APPLICATION OF "=" TO List_Cell_Pointer_Type OPERANDS STILL PERFORMS ORDINARY COMPARISONS OF ACCESS VALUES.

(THE EXPLICIT DECLARATION OF A FUNCTION "=" FOR List_Type OVERRIDES THE STANDARD EQUALITY OPERATION THAT List_Type IMPLICITLY DERIVES FROM List_Cell_Pointer_Type.

IN THE "=" FUNCTION BODY, TYPE CONVERSIONS FROM THE DERIVED TYPE TO THE PARENT TYPE ARE USED TO GIVE Left_Cell_Pointer AND Right_Cell_Pointer VALUES OF List_Cell_Pointer_Type THAT CORRESPOND TO THE VALUES OF THE List_Type PARAMETERS. NOW THE OCCURRENCES OF = WITHIN THE FUNCTION BODY HAVE List_Cell_Pointer_Type OPERANDS, SO THEY INVOKE THE STANDARD VERSION OF EQUALITY.

List_Type CORRESPONDS TO THE HIGH-LEVEL VIEW OF THE LIST AS A SEQUENCE OF INTEGERS, AND ITS VERSION OF EQUALITY COMPARES SEQUENCES OF INTEGERS. List_Cell_Pointer_Type CORRESPONDS TO THE LOW-LEVEL VIEW OF THE LIST AS A LINKED LIST DATA STRUCTURE, AND ITS VERSION OF EQUALITY COMPARES THE VALUES USED AS LINKS.

USES OF DERIVED TYPES -- MULTIPLE VERSIONS OF OPERATORS (Continued)

- **SOLUTION:** DEFINE A LINKED LIST ACCESS TYPE FOR INTERNAL USE ONLY AND DERIVE List_Type FROM THAT TYPE.

  List_Type AND ITS PRIVATE TYPE CAN HAVE DIFFERENT VERSIONS OF =.

```
package List_Package is
   type List_Type is limited private;
   ...
   function "=" (Left, Right : List_Type) return Boolean;
   ...
private
   type List_Cell_Type;
   type List_Cell_Pointer_Type is access List_Cell_Type;
   type List_Cell_Type is
      record
         Integer_Part : Integer;
         Link_Part    : List_Cell_Pointer_Type;
      end record;
   type List_Type is new List_Cell_Pointer_Type;
end List_Package;
```

INSTRUCTOR NOTES

USES OF DERIVED TYPES -- MULTIPLE VERSIONS OF OPERATORS (Continued)

```
package body List_Package is
   ...
   function "=" (Left, Right : List_Type) return Boolean is
      Left_Cell_Pointer  : List_Cell_Pointer_Type := List_Cell_Pointer_Type(Left);
      Right_Cell_Pointer : List_Cell_Pointer_Type := List_Cell_Pointer_Type(Right);
   begin -- "="
      ...
      if Right_Cell_Pointer = null ...           -- ORDINARY "="
      ...
      return Right_Cell_Pointer = null;          -- ORDINARY "="
   end "=";
end List_Package;
```

- List_Type AND List_Cell_Pointer_Type CORRESPOND TO DIFFERENT VIEWS OF THE LINKED LIST, AT DIFFERENT LEVELS OF ABSTRACTION. THERE IS A DIFFERENT VIEW OF EQUALITY AT EACH LEVEL OF ABSTRACTION.

INSTRUCTOR NOTES

STUDENTS WILL SOON BE GIVEN AN EXERCISE TO WRITE A GENERIC PACKAGE LIKE List_Package_Template. AFTERWARDS, THE USE OF THIS GENERIC PACKAGE TO IMPLEMENT A LINKED STACK WILL BE EXPLAINED IN GREATER DETAIL.

LAST BULLET: A SUBTYPE DECLARATION CAN DECLARE A SYNONYM FOR A TYPE OR SUBTYPE NAME, BUT IT CANNOT ACT AS THE FULL DECLARATION OF A PRIVATE TYPE.

USES OF DERIVED TYPES -- PRIVATE TYPES IMPLEMENTED BY GENERIC INSTANTIATION

```
package Integer_Stack_Package is
   type Integer_Stack_Type is private;
   ...

private

   type Integer_List_Package is new
      List_Package_Template (Element_Type =>  Integer);
   type Integer_Stack_Type is new Integer_List_Package.List_Type;

end Integer_Stack_Package;
```

- THE PRIVATE TYPE DECLARATION MUST BE MATCHED BY A FULL TYPE DECLARATION IN THE PRIVATE PART.

- THERE IS NO FORM OF type DECLARATION THAT CAN DECLARE Integer_Stack_Type TO BE A SYNONYM FOR Integer_List_Package.List_Type. INSTEAD, WE DECLARE Integer_Stack_Type TO BE DERIVED FROM Integer_List_Package.List_Type.

INSTRUCTOR NOTES

THE RECORD TYPE DECLARATION FOR Old_Format GIVES AN ABSTRACT VIEW OF THE DATA IN THE OLD AND NEW FILES. IT DOES NOT SPECIFY THE ORDER IN WHICH RECORD COMPONENTS ARE STORED INTERNALLY.

THE REPRESENTATION CLAUSES SPECIFY PHYSICAL VIEWS OF THE DATA, ASSIGNING RECORD COMPONENTS TO SPECIFIC RANGES OF BITS. THE REPRESENTATION CLAUSE FOR Old_Format SPECIFIES THE OLD ARRANGEMENT OF THE DATA, WHILE THAT FOR New_Format SPECIFIES THE NEW ARRANGEMENT.

THE TYPE CONVERSION FROM THE PARENT TYPE TO THE DERIVED TYPE INSIDE THE CALL ON WRITE DOES ALL THE WORK OF BUILDING NEW-STYLE RECORDS OUT OF OLD-STYLE RECORDS

REPRESENTATION CLAUSES ARE COVERED IN PART 7 OF THE COURSE. WE WILL REVIEW THIS USE OF DERIVED TYPES THERE.

USES OF DERIVED TYPES -- MULTIPLE REPRESENTATIONS

- REPRESENTATION CLAUSES (TO BE DISCUSSED LATER) ALLOW A PROGRAMMER TO SPECIFY THE INTERNAL REPRESENTATION OF DATA.

- A PARENT TYPE AND DERIVED TYPE NEED NOT HAVE THE SAME INTERNAL REPRESENTATION. WHEN THEY DON'T, TYPE CONVERSION ENTAILS A CHANGE IN REPRESENTATION.

- EXAMPLE: CONVERT A FILE WITH RECORDS OF THE FORM

account #	name	address

TO ONE WITH THE FORM

name	address	account #

INSTRUCTOR NOTES

USES OF DERIVED TYPES -- MULTIPLE REPRESENTATIONS (Continued)

```
with Sequential_IO;

procedure Convert_File is
   type Old_Format is
      record
         Account_Number : Integer range 111_111 .. 999_999;
         Name, Address  : String (1 .. 80);
      end record;
   type New_Format is new Old_Format;
   for Old_Format use ...;                        -- REPRESENTATION CLAUSE
   for New_Format use ...;                        -- REPRESENTATION CLAUSE
   Input_Record : Old_Format;
   package Old_Format_IO is new Sequential_IO (Old_Format);
   package New_Format_IO is new Sequential_IO (New_Format);
   use Old_Format_IO, New_Format_IO;
   Input_File  : Old_Format_IO.File_Type;
   Output_File : New_Format_IO.File_Type;
begin -- Convert_File
   Open (Input_File, In_File, "OLDDATA.DAT");
   Open (Output_File, Out_File, "NEWDATA.DAT");
   while not End_Of_File (Input_File) loop
      Read (Input_File, Input_Record);
      Write (Output_File, New_Format (Input_Record));     -- TYPE CONVERSION
   end loop;
   Close (Input_File);
   Close (Output_File);
end Convert_File;
```

INSTRUCTOR NOTES

THE NEXT SLIDE GIVES A SPECIFIC EXAMPLE. EXCEPT FOR THE FACT THAT CONSTANTS ARE NOT DERIVED, COMPLICATIONS USUALLY ARISE ONLY WHEN THE PARENT TYPE IS NOT THE ONLY TYPE DERIVED IN ITS PACKAGE.

CAVEAT

- WHEN THE PARENT TYPE IS PROVIDED BY A PACKAGE, ONLY SOME OF THE THINGS PROVIDED BY THE PACKAGE ARE DERIVED

  -- THE TYPE ITSELF

  -- SUBPROGRAMS WITH PARAMETERS OR RESULTS BELONGING TO THE TYPE

- THE FOLLOWING ARE NOT DERIVED

  -- CONSTANTS OF THE TYPE

  -- OTHER TYPES PROVIDED BY THE PACKAGE

  -- SUBPROGRAMS PROVIDED BY THE PACKAGE THAT DO NOT HAVE PARAMETERS OR RESULTS OF THE DERIVED TYPE

- TO PRESERVE YOUR SANITY:

  FOLLOW THE DERIVED TYPE DECLARATION BY OTHER DECLARATIONS THAT ALLOW YOU TO USE THE NON-DERIVED ENTITIES AS IF THEY WERE DERIVED.

INSTRUCTOR NOTES

Matrix_Package PROVIDES THREE TYPES: A TYPE FOR MATRICES CONTAINING Float COMPONENTS, A TYPE WHOSE VALUES SPECIFY ROWS OF SUCH A MATRIX, AND A TYPE WHOSE VALUES SPECIFY COLUMNS OF SUCH A MATRIX. GIVEN A MATRIX, First_Row RETURNS THE SPECIFICATION OF THE MATRIX'S FIRST ROW. GIVEN THE SPECIFICATION OF ONE ROW, Row_After RETURNS THE SPECIFICATION OF THE NEXT ROW. First_Column AND Column_After WORK ANALOGOUSLY.

Table_Type IS DERIVED FROM Matrix_Package.Matrix_Type. BECAUSE First_Row AND First_Column HAVE PARAMETERS IN THE PARENT TYPE, VERSIONS OF First_Row AND First_Column TAKING Table_Type PARAMETERS ARE ALSO DERIVED. IT WOULD BE AN ERROR TO CALL Matrix_Package.First_Row WITH A Table_Type PARAMETER, SINCE THAT VERSION OF First_Row ONLY WORKS WITH Matrix_Type PARAMETERS. HOWEVER, IF NOTHING ELSE IS DONE, THE NON-DERIVED FACILITIES OF Matrix_Package WILL HAVE TO BE NAMED WITH NAMES OF THE FORM "Matrix_Package.________".

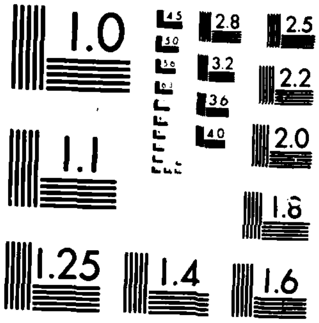
ANSWERS: (a) YES, (b) NO, (c) YES, (d) NO, (e) NO, (f) NO

THE NEXT SLIDE SHOWS HOW TO LIMIT CONCERN ABOUT WHICH ENTITIES ARE AND ARE NOT INHERITED.

AD-A165 076 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506

4/7

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

EXAMPLE

```
package Matrix_Package is
   type Matrix_Type is private;
   type Row_Name_Type is private;
   type Column_Name_Type is private;
   function First_Row (Matrix : Matrix_Type) return Row_Name_Type;
   function Row_After (Row : Row_Name_Type) return Row_Name_Type;
   function First_Column (Matrix : Matrix_Type) return Column_Name_Type;
   function Column_After (Column : Column_Name_Type) return Column_Name_Type;
   function Element_At (Row : Row_Name_Type; Column : Column_Name_Type) return Float;
   ...
   Zero_Matrix : constant Matrix_Type;
private
   ...
end Matrix_Package;
   ...
type Table_Type is new Matrix_Package.Matrix_Type;
```

WHICH OF THE FOLLOWING ARE INHERITED?

		YES	NO
(a)	First_Row		
(b)	Row_After		
(c)	First_Column		
(d)	Column_After		
(e)	Element_At		
(f)	Zero_Matrix		

INSTRUCTOR NOTES

TO AVOID THE NECESSITY TO KEEP TRACK OF WHICH ENTITIES ARE AND ARE NOT DERIVED, WE IMMEDIATELY USE SUBTYPE AND RENAMING DECLARATIONS TO ALLOW THE NON-DERIVED FACILITIES TO BE REFERRED TO IN THE SAME WAY AS THE DERIVED FACILITIES -- BY THEIR SIMPLE NAMES. IN THE CASE OF A CONSTANT, SOMETHING MORE IS NECESSARY -- WE MUST CONVERT THE VALUE OF THE PARENT TYPE CONSTANT TO THE CORRESPONDING VALUE IN THE DERIVED TYPE.

FORTUNATELY, SITUATIONS AS COMPLICATED AS THIS ARISE INFREQUENTLY SINCE PACKAGES OF TEN PROVIDE A SINGLE DERIVED TYPE AND SUBPROGRAMS THAT ALL HAVE PARAMETERS OR RESULTS OF THAT TYPE. UNFORTUNATELY, THE EXAMPLES AND LAB EXERCISES THAT ARE COMING UP DO ENTAIL SITUATIONS LIKE THIS.

MANAGING THE PROBLEM

```
-- DECLARATIONS FOR NON-DERIVED ENTRIES IN Matrix_Package;

subtype Row_Name_Type is Matrix_Package.Row_Name_Type;
subtype Column_Name_Type is Matrix_Package.Column_Name_Type;
function Row_After (Row : Row_Name_Type) return Row_Name_Type
   renames Matrix_Package.Row_After;
function Column_After (Column : Column_Name_Type) return Column_Name_Type
   renames Matrix_Package.Column_Name_Type;
function Element_At (Row : Row_Name_Type; Column : Column_Name_Type) return Float
   renames Matrix_Package.Element_At;
Zero_Table : constant Table_Type := Table_Type (Matrix_Package.Zero_Matrix);
```

INSTRUCTOR NOTES

SECTION 13

UNCHECKED DEALLOCATION

INSTRUCTOR NOTES

DEPENDING ON THE APPLICATION AND THE RUNTIME SUPPORT ENVIRONMENT, THE FINITENESS OF STORAGE MAY OR MAY NOT BE A SERIOUS CONCERN.

THUS NOT ALL PROGRAMS NEED WORRY ABOUT DEALLOCATION.

DEALLOCATION OF ALLOCATED VARIABLES

- THE STORAGE AVAILABLE FOR ALLOCATED VARIABLES IS FINITE.
- EVALUATION OF AN ALLOCATOR WHEN NO STORAGE IS LEFT RAISES THE PREDEFINED EXCEPTION Storage_Error.
- ADA PROVIDES A MEANS FOR DEALLOCATING A VARIABLE -- THAT IS, RECYCLING ITS STORAGE SO THAT THE STORAGE MAY BE RE-USED.

INSTRUCTOR NOTES

UNPREDICTABLE EFFECTS CAN BE FATAL, E.G., A BRANCH TO AN ADDRESS WHOSE CONTENTS WERE NOT MEANT TO BE INTERPRETED AS AN INSTRUCTION.

AN EXAMPLE OF SEVERAL ACCESS TYPE OBJECTS DESIGNATING A VARIABLE TO BE DEALLOCATED WILL FOLLOW THE NEXT SLIDE.

DEALLOCATION CAN BE DANGEROUS

- ONCE YOU HAVE DEALLOCATED AN ALLOCATED VARIABLE, YOU MUST NO LONGER USE IT.

  -- ALL OR PART OF ITS STORAGE MAY BE REUSED FOR OTHER PURPOSES (SUCH AS ALLOCATING NEW VARIABLES, SAVING THE RETURN ADDRESS FOR A SUBPROGRAM CALL, ETC.)

  -- THE CONTENTS OF THE STORAGE MAY CHANGE UNPREDICTABLY AFTER DEALLOCATION.

  -- MODIFYING THAT STORAGE CAN HAVE UNPREDICTABLE EFFECTS ON A COMPUTATION.

- PREVENTING FURTHER USE OF THE VARIABLE CAN BE TRICKY WHEN SEVERAL ACCESS TYPE OBJECTS DESIGNATE THE VARIABLE TO BE DEALLOCATED.

INSTRUCTOR NOTES

BULLET 1: THE REASON FOR THE NAME Unchecked_Deallocation IS GIVEN IN A LATER SLIDE. DEFER THIS ISSUE FOR NOW.

BULLET 2: THE <u>with</u> CLAUSE MAKES THE TEMPLATE AVAILABLE IN THIS COMPILATION UNIT FOR INSTANTIATION. THE INSTANTIATION CREATES AN INSTANCE FOR THIS PARTICULAR ACCESS TYPE.

BULLET 3: SETTING THE ACTUAL PARAMETER TO <u>null</u> IS A PARTIAL SAFEGUARD. IT PREVENTS SUBSEQUENT USE OF THE ACTUAL PARAMETER, BUT NOT OF OTHER ACCESS TYPE OBJECTS WITH THE SAME VALUE.

BULLET 4: THE PROGRAMMER IS RESPONSIBLE FOR NOT USING THESE OTHER OBJECTS.

THE GENERIC PROCEDURE Unchecked_Deallocation

- DEALLOCATION IS PERFORMED BY INSTANTIATIONS OF THE PREDEFINED GENERIC PROCEDURE Unchecked_Deallocation.

```
generic
   type Object is limited private;
   type Name is access Object;
procedure Unchecked_Deallocation (X : in out Name);
```

- TO DEALLOCATE VARIABLES, YOU MUST

  -- NAME THE GENERIC TEMPLATE IN A <u>with</u> CLAUSE

  ```
  with Unchecked_Deallocation;
  ```

  -- INSTANTIATE THE GENERIC PROCEDURE

  ```
  procedure name of instantiation is new
     Unchecked_Deallocation
        (Object => name of designated type,
         Name   => name of access type );
  ```

  -- CALL THE INSTANTIATION USING A VARIABLE OF THE NAMED ACCESS TYPE AS A PARAMETER

- THE CALL DEALLOCATES THE VARIABLE DESIGNATED BY THE ACTUAL PARAMETER AND SETS THE ACTUAL PARAMETER TO null. (IF THE ACTUAL PARAMETER CONTAINED null TO BEGIN WITH, THE CALL HAS NO EFFECT.)

- DANGER! OTHER ACCESS VARIABLES THAT CONTAINED THE SAME ACCESS VALUE AS THE PARAMETER ARE UNAFFECTED. THEY NOW DESIGNATE RECYCLED STORAGE.

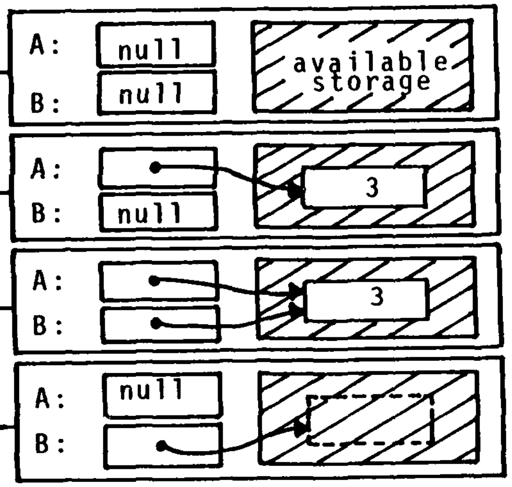
INSTRUCTOR NOTES

IN THE CALL ON Free_Allocated_Integer, THE X IS THE FORMAL PARAMETER NAME GIVEN IN THE GENERIC DECLARATION (SEE PREVIOUS SLIDE).

THE CALL HAS TWO EFFECTS. A IS SET TO null AND THE STORAGE FOR THE ALLOCATED VARIABLE A ORIGINALLY DESIGNATED IS MADE AVAILABLE FOR RE-USE. THUS B NOW DESIGNATES A PIECE OF AVAILABLE STORAGE THAT MAY LATER BE USED, IN WHOLE OR IN PART, FOR SOME OTHER PURPOSE.

EXAMPLE OF DEALLOCATION

```
with Unchecked_Deallocation;

procedure Example is

   type Integer_Pointer is access Integer;
   A, B : Integer_Pointer;
   procedure Free_Allocated_Integer is new
      Unchecked_Deallocation (Object =>  Integer, Name =>  Integer_Pointer);

begin
----------------------------------------

   A := new Integer'(3);
----------------------------------------

   B := A;
----------------------------------------

   Free_Allocated_Integer (X =>  A);
----------------------------------------

   ~~B.all := 4;~~  -- ERRONEOUS AND PROBABLY SUICIDAL

end Example;
```



INSTRUCTOR NOTES

```
with Unchecked_Deallocation;


procedure Free_String is new
   Unchecked_Deallocation (Object => String, Name => String_Pointer_Type);


-- -- -- -- --

Free_String (S);
```

NOTE THE with CLAUSE.

EXERCISE

ASSUMING THE DECLARATIONS

```
type String_Pointer_Type is access String;
S : String_Pointer_Type := new String'("HELLO!");
```

WRITE A SEPARATELY COMPILED GENERIC INSTANTIATION AND A PROCEDURE CALL TO DEALLOCATE THE VARIABLE POINTED TO BY S.

INSTRUCTOR NOTES

THIS MEANS THAT UNCHECKED DEALLOCATION SHOULD ONLY BE USED WHEN IT IS REALLY NECESSARY AND WHEN THE PROGRAMMER HAS TAKEN SPECIAL PRECAUTIONS.

WHY IT'S CALLED "UNCHECKED" DEALLOCATION

- NORMALLY, ADA'S RULES PREVENT AN ACCESS VALUE FROM POINTING TO "NOWHERE."

  -- ALL ACCESS VARIABLES ARE INITIALIZED TO NULL.

  -- NEW ACCESS VALUES ONLY ARISE FROM EVALUATION OF ALLOCATORS

- WHEN THE PROGRAMMER DEALLOCATES STORAGE, THERE IS NO WAY TO CHECK FOR THE USE OF A NOW-MEANINGLESS ACCESS VALUE.

INSTRUCTOR NOTES

PART V. APPLICATIONS

14. GENERIC PACKAGES FOR STACKS

15. TREES

16. SEARCHING

17. SORTING

18. LINKED LIST IMPLEMENTATION OF SETS

19. MERGEABLE SETS

20. GRAPHS

INSTRUCTOR NOTES

SECTION 14

GENERIC PACKAGES FOR STACKS

INSTRUCTOR NOTES

REASONS FOR MAKING THE STACK TYPE LIMITED PRIVATE WILL BE GIVEN LATER.

THE FIRST VERSION IS LIKELY TO REQUIRE LESS SPACE AND MAY ALSO BE FASTER (THOUGH THIS IS NOT CLEAR).

THE SECOND VERSION USES MORE STORAGE, BUT ALLOWS IT TO BE USED MORE FLEXIBLY. IT DOES NOT PREALLOCATE A FIXED AMOUNT OF SPACE TO EACH STACK, BUT ALLOWS MANY STACKS TO GROW QUITE LARGE AS LONG AS THEY ARE NOT LARGE AT THE SAME TIME. (THAT IS, THE ONLY LIMITING FACTOR IS THE TOTAL SIZE OF ALL STACKS AT A GIVEN MOMENT.)

TWO GENERIC PACKAGES FOR STACK TYPES

GENERIC PARAMETER IS THE TYPE OF THE ELEMENTS TO BE STACKED. THE STACK TYPE WILL BE LIMITED PRIVATE.

VERSION 1: THE STACK TYPE WILL HAVE A DISCRIMINANT GIVING THE MAXIMUM CAPACITY OF A GIVEN STACK.

VERSION 2: THERE IS NO SPECIFIED BOUND ON STACK CAPACITY.

VERSION 1 WILL BE IMPLEMENTED WITH ARRAYS, VERSION 2 WITH LINKED LISTS.

INSTRUCTOR NOTES

POINT OUT THAT THE DISCRIMINANT APPEARS IN BOTH THE PRIVATE TYPE DECLARATION AND THE FULL DECLARATION.

SINCE Stack_Type IS LIMITED PRIVATE, ONE Stack_Type VALUE CANNOT BE COPIED TO ANOTHER. HENCE THERE IS NO NEED FOR UNCONSTRAINED Stack_Type OBJECTS. HENCE THE DISCRIMINANT IS NOT GIVEN A DEFAULT INITIAL VALUE. ALL OBJECTS IN THE TYPE MUST BE DECLARED WITH DISCRIMINANT CONSTRAINTS.

GENERIC DECLARATION FOR VERSION 1

```
generic

   type Element_Type is private;

package Stack_Package_Template is

   type Stack_Type (Capacity : Positive) is limited private;
   procedure Reset_Stack (Stack : in out Stack_Type);
   function Is_Empty (Stack : Stack_Type) return Boolean;
   function Is_Full (Stack : Stack_Type) return Boolean;
   procedure Push (Element : in Element_Type; Stack : in out Stack_Type);
   procedure Pop (Element : out Element_Type; Stack : in out Stack_Type);
   Stack_Overflow, Stack_Underflow : exception;

private

   type Element_List_Type is array (Positive range < > ) of Element_Type;
   type Stack_Type (Capacity : Positive) is
      record
         Top_Part          : Natural := 0;
         Element_List_Part : Element_List_Type (1 .. Capacity);
      end record;

end Stack_Package_Template;
```

POINT OUT THE USE OF THE DISCRIMINANT Capacity AS AN ORDINARY RECORD COMPONENT IN Is_Full AND Push.

ANSWERS:

```
function Is_Empty (Stack : Stack_Type) return Boolean is
begin
   return Stack.Top_Part = 0;
end Is_Empty;

function Is_Full (Stack : Stack_Type) return Boolean is
begin
   return Stack.Top_Part = Stack.Capacity;
end Is_Full;

procedure Pop (Element : out Element_Type; Stack : in out Stack_Type) is
begin
   if Stack.Top_Part = 0 then
      raise Stack_Underflow;
   else
      Element := Stack.Element_List_Part (Stack.Top_Part);
      Stack.Top_Part := Stack.Top_Part - 1;
   end if;
end Pop;
```

GENERIC BODY FOR VERSION 1

```
package body Stack_Package_Template is
   procedure Reset_Stack (Stack : in out Stack_Type) is
   begin
      Stack.Top_Part := 0;
   end Reset_Stack;

   function Is_Empty (Stack : Stack_Type) return Boolean is
   begin

   end Is_Empty;

   function Is_Full (Stack : Stack_Type) return Boolean is
   begin

   end Is_Full;

   procedure Push (Element : in Element_Type; Stack : in out Stack_Type) is
   begin
      if Stack.Top_Part = Stack.Capacity then
         raise Stack_Overflow;
      else
         Stack.Top_Part := Stack.Top_Part + 1;
         Stack.Element_List_Part (Stack.Top_Part) := Element;
      end if;
   end Push;

   procedure Pop (Element : out Element_Type; Stack : in out Stack_Type) is
   begin





   end Pop;
end Stack_Package_Template;
```

INSTRUCTOR NOTES

IN THIS VERSION, Stack_Type DOES NOT HAVE A DISCRIMINANT.

THE EXCEPTION Stack_Overflow NOW HAS A DIFFERENT MEANING. RATHER THAN INDICATING THAT A PARTICULAR STACK IS FILLED TO CAPACITY, IT INDICATES THAT NO STORAGE IS LEFT FOR PUSHING AN ITEM ONTO ANY STACK.

THE NEW FUNCTION Stack_Space_*Available TELLS* WHETHER Push CAN BE CALLED AT A GIVEN TIME WITHOUT RAISING Stack_Overflow. THERE IS NO OTHER WAY TO DETERMINE THIS WITHOUT ACTUALLY TRYING THE CALL. IN GENERAL, WHEN A SUBPROGRAM CALL MIGHT RAISE AN EXCEPTION, IT IS GOOD TO PROVIDE THE USER WITH A WAY TO DETERMINE THIS WITHOUT ACTUALLY CALLING THE SUBPROGRAM.

ALL Stack_Type OBJECTS ARE EXPLICITLY INITIALIZED TO NULL, REPRESENTING THE EMPTY STACK.

BECAUSE THE PRIVATE TYPE DECLARATION ACTS AS A FORWARD REFERENCE, NO INCOMPLETE TYPE DECLARATION IS NECESSARY TO DEFINE THIS RECURSIVE TYPE.

GENERIC DECLARATION FOR VERSION 2

```
generic
   type Element_Type is private;
package Stack_Package_Template is
   type Stack_Type is limited private;
   procedure Reset_Stack (Stack : in out Stack_Type);
   function Is_Empty (Stack : Stack_Type) return Boolean;
   function Stack_Space_Available return Boolean;
   procedure Push (Element : in Element_Type; Stack : in out Stack_Type);
   procedure Pop (Element : out Element_Type; Stack : in out Stack_Type);
   Stack_Overflow, Stack_Underflow : exception;
private
   type Stack_Cell_Type is
      record
         Element_Part      : Element_Type;
         Link_Part         : Stack_Type;
      end record;
   type Stack_Type is access Stack_Cell_Type;
end Stack_Package_Template;
```

INSTRUCTOR NOTES

THE VARIABLE Next_Cell DESIGNATES THE NEXT STACK CELL THAT WILL BE USED DURING A Push, OR HOLDS THE VALUE null IF NO MORE CELLS ARE AVAILABLE. THIS SIMPLIFIES THE IMPLEMENTATION OF Stack_Space_Available BY "LOOKING AHEAD" ONE ALLOCATION.

THE PROCEDURE Attempt_Next_Cell_Allocation IS CALLED TO ADVANCE Next_Cell TO ITS NEXT APPROPRIATE VALUE. IT TRIES TO ALLOCATE A NEW VARIABLE. IF THIS ATTEMPT SUCCEEDS, Next Cell IS SET TO DESIGNATE THE NEW VARIABLE. IF THE ATTEMPT RAISES Storage_Error, THE HANDLER SETS Next_Cell to null.

POINT OUT THE INSTANTIATION OF Unchecked_Deallocation.

THE SUBUNITS FOLLOW ON SUBSEQUENT SLIDES.

GENERIC BODY FOR VERSION 2

```
with Unchecked_Deallocation;

package body Stack_Package_Template is
   Next_Cell : Stack_Type := new Stack_Cell_Type;
   -- POINTS TO THE CELL THAT WILL BE ADDED TO A STACK BY THE NEXT CALL ON PUSH
   -- EQUALS null WHEN NO CELLS ARE AVAILABLE.

   procedure Recycle_Stack_Cell is new
      Unchecked_Deallocation (Stack_Cell_Type, Stack_Type);

   procedure Attempt_Next_Cell_Allocation is
   begin
      Next_Cell:= new Stack_Cell_Type;
   exception
      when Storage_Error =>
         Next_Cell := null;
   end Attempt_Next_Cell_Allocation;

   procedure Reset_Stack (Stack : in out Stack_Type) is separate;
   function Is_Empty (Stack : Stack_Type) return Boolean is separate;
   function Stack_Space_Available return Boolean is separate;
   procedure Push (Element : in Element_Type; Stack : in out Stack_Type)
      is separate;
   procedure Pop (Element : out Element_Type; Stack : in out Stack_Type)
      is separate;

end Stack_Package_Template;
```

INSTRUCTOR NOTES

THE ASSIGNMENT "Stack := null;" WOULD INSTANTANEOUSLY RESET THE STACK, BUT IT WOULD NOT RECYCLE THE STORAGE USED BY THE STACK. INSTEAD, WE DEALLOCATE THE CELLS THAT WERE ON THE STACK.

WE KNOW THE DEALLOCATION IS SAFE BECAUSE Stack_Type IS LIMITED PRIVATE. THUS THE ACTUAL PARAMETER IS THE ONLY OBJECT THAT COULD HAVE BEEN DESIGNATING THE "TOP" STACK CELL BEFORE THE CALL.

RECYCLING STORAGE MAY CURE A PREVIOUS OUT-OF-STORAGE CONDITION. THUS, IF Next_Cell WAS PREVIOUSLY null, WE CALL Attempt_Next_Cell_Allocation, WHICH CAN REALLOCATE ONE OF THE DEALLOCATED CELLS AND SET Next_Cell TO DESIGNATE IT.

Reset_Stack SUBUNIT

```
separate (Stack_Package_Template)

procedure Reset_Stack ( Stack : in out Stack_Type) is
   Old_Cell : Stack_Type;
begin -- Reset_Stack
   while Stack /= null loop
      Old_Cell := Stack;
      Stack := Stack.Link_Part;
      Recycle_Stack_Cell (Old_Cell);
   end loop;
   -- Stack = null
   if Next_Cell = null then
      Attempt_Next_Cell_Allocation;
   end if;
end Reset_Stack;
```

ANSWERS:

```
separate (Stack_Package_Template)

function Is_Empty (Stack : Stack_Type) return Boolean is
begin
  return Stack = null;
end Is_Empty;

-----------------------------------------------------------------------------

separate (Stack_Package_Template)

function Stack_Space_Available return Boolean is
begin
  return Next_Cell /= null;
end Stack_Space_Available;
```

MORE STACK FUNCTIONS

```
separate (Stack_Package_Template)

function Is_Empty (Stack : Stack_Type) return Boolean is
begin

end Is_Empty;

-----------------------------------------------------------------------

separate (Stack_Package_Template)

function Stack_Space_Available return Boolean is
begin

end Stack_Space_Available;
```

INSTRUCTOR NOTES

IN THE NORMAL CASE, THIS PROCEDURE ADDS THE CELL DESIGNATED BY Next_Cell TO THE FRONT OF THE LIST, THEN RESETS Next_Cell BY CALLING Attempt_Next_Cell_Allocation.

PROCEDURE PUSH

```
separate (Stack_Package_Template)

procedure Push (Element : in Element_Type; Stack : in out Stack_Type) is
begin
   if Next_Cell = null then
      raise Stack_Overflow;
   else
      Next_Cell.all := (Element_Part =>  Element, Link_Part =>  Stack);
      Stack := Next_Cell;
      Attempt_Next_Cell_Allocation;
   end if;
end Push;
```

INSTRUCTOR NOTES

THE ONE CELL BEING REMOVED IS RECYCLED. IF Next_Cell = null, THEN NO FURTHER CELLS WERE AVAILABLE FOR PUSHES. WE SET Next_Cell TO DESIGNATE THE POPPED CELL, SO THAT IT WILL BE AVAILABLE FOR THE NEXT PUSH. IF Next_Cell /= null, WE DEALLOCATE THE CELL, MAKING IT AVAILABLE FOR FUTURE ALLOCATION.

AS WITH Reset_Stack, WE KNOW DEALLOCATION IS SAFE BECAUSE Stack_Type IS LIMITED PRIVATE, AND THE ONLY OBJECT DESIGNATING A STACK CELL IS A Stack_Type VARIABLE (WHEN THE CELL IS AT THE TOP OF THE STACK) OR THE NEXT-HIGHER CELL IN THE STACK (WHEN THE CELL IS NOT AT THE TOP).

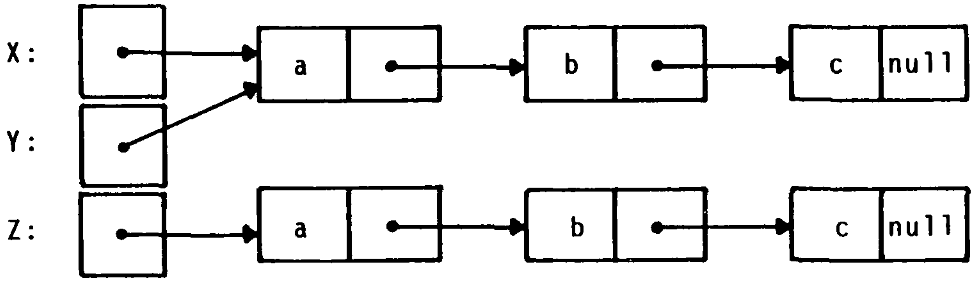
PROCEDURE POP

```
separate (Stack_Package_Template)

procedure Pop (Element : out Element_Type; Stack : in out Stack_Type) is
   Old_Cell : Stack_Type;
begin -- Pop
   if Stack = null then
      raise Stack_Underflow;
   else
      Old_Cell := Stack;
      Element := Stack.Element_Part;
      Stack := Stack.Link_Part;
      if Next_Cell = null then
         Next_Cell := Old_Cell;
      else
         Recycle_Stack_Cell (Old_Cell);
      end if;
   end if;
end Pop;
```

INSTRUCTOR NOTES

ASSIGNMENT AND EQUALITY ARE NOT TYPICAL OPERATIONS IN STACKS ANYWAY.

IF NECESSARY, DRAW A PICTURE TO SHOW WHY PREDEFINED "=" FOR THE LIST VERSION WOULD HAVE BEEN MISLEADING.



FROM AN ABSTRACT POINT OF VIEW, X = Y = Z. IN TERMS OF POINTER EQUALITY, X = Y, BUT X /= Z AND Y /= Z.

WHY WAS Stack_Type LIMITED PRIVATE?

FOR DIFFERENT REASONS IN EACH VERSION.

FOR VERSION 1: PREDEFINED EQUALITY WOULD HAVE BEEN MISLEADING, BECAUSE IT WOULD HAVE COMPARED ALL COMPONENTS OF THE ARRAY, INCLUDING THOSE BEYOND THE TOP OF THE STACK.

FOR VERSION 2: ASSIGNMENT WOULD HAVE CAUSED STACK CELLS TO BE SHARED BY TWO DIFFERENT STACK VARIABLES. THIS WOULD HAVE MADE A POP OF ONE STACK DEALLOCATE A CELL ON ANOTHER STACK.

ALSO, PREDEFINED EQUALITY WOULD TEST FOR THE SAME ACCESS VALUE RATHER THAN FOR LISTS WITH THE SAME CONTENTS.

INSTRUCTOR NOTES

THE STACK OBJECT IS EMBODIED IN VARIABLE DECLARATIONS INSIDE THE PACKAGE BODY.

THE FOLLOWING SLIDES ILLUSTRATE THIS APPROACH FOR THE BOUNDED STACK ONLY.

AN ALTERNATIVE APPROACH

- A GENERIC PACKAGE HIDING A SINGLE STACK OBJECT.
- ALL OPERATIONS PROVIDED BY THE PACKAGE OPERATE ON THIS ONE STACK. THUS THERE ARE NO Stack_Type PARAMETERS.
- FOR THE BOUNDED STACK, THE STACK CAPACITY IS PROVIDED AS A GENERIC PARAMETER.
- TO CREATE MULTIPLE STACKS, INSTANTIATE THE PACKAGE SEVERAL TIMES (WITH IDENTICAL OR DIFFERENT PARAMETERS).

INSTRUCTOR NOTES

ONLY THE ARRAY IMPLEMENTATION IS SHOWN. FOR LINKED STACKS, THE SECOND GENERIC PARAMETER WOULD BE ABSENT AND Stack_Overflow WOULD HAVE THE SAME MEANING AS FOR LINEAR STACKS (NO MORE STORAGE AVAILABLE FOR THIS STACK).

POINT OUT THE ABSENCE OF Stack_Type PARAMETERS.

Pop IS MADE A *PROCEDURE RATHER THAN A FUNCTION BECAUSE IT HAS* A SIDE-EFFECT OF CHANGING THE STACK. A PROCEDURE CALL STATEMENT LOOKS LIKE A COMMAND TO CHANGE THE STATE OF A COMPUTATION, WHILE A FUNCTION CALL EXPRESSION LOOKS LIKE THE PASSIVE DESCRIPTION OF A VALUE. THEREFORE, A PROCEDURE MAKES THE EFFECT OF THE PROGRAM MORE APPARENT.

SINGLE-OBJECT GENERIC PACKAGE DECLARATION FOR BOUNDED STACKS

```
generic

   type Element_Type is private;
   Capacity : in Positive;

package Stack_Object_Template is

   procedure Reset_Stack;
   function Is_Empty return Boolean;
   function Is_Full return Boolean;
   procedure Push (Element : in Element_Type);
   procedure Pop (Element : out Element_Type);
   Stack_Overflow, Stack_Underflow : exception;

end Stack_Object_Template;
```

WHAT WAS PREVIOUSLY A DISCRIMINANT IS NOW A GENERIC FORMAL CONSTANT. WHAT WERE PREVIOUSLY ORDINARY RECORD COMPONENTS ARE NOW VARIABLES DECLARED IN THE PACKAGE BODY.

ANSWERS:

```
procedure Reset_Stack is
begin
   Top := 0;
end Reset_Stack;

function Is_Empty return Boolean is
begin
   return Top = 0;
end Is_Empty;

function Is_Full return Boolean is
begin
   return Top = Capacity;
end Is_Full;
```

BODY FOR SINGLE-OBJECT GENERIC STACK PACKAGE

```
package body Stack_Object_Template is

   Element_List : array (1 .. Capacity) of Element_Type;
   Top          : Integer range 0 .. Capacity;

   procedure Reset_Stack is
   begin

   end Reset_Stack;

   function Is_Empty return Boolean is
   begin

   end Is_Empty;

   function Is_Full return Boolean is
   begin

   end Is_Full;

   procedure Push (Element : in Element_Type) is separate;

   procedure Pop (Element : out Element_Type) is separate;

end Stack_Object_Template;
```

INSTRUCTOR NOTES

SUBUNITS FOR Push AND Pop

```
separate (Stack_Object_Template)

procedure Push (Element : in Element_Type) is
begin
   if Top = Capacity then
      raise Stack_Overflow;
   else
      Top := Top + 1;
      Element_List (Top) := Element;
   end if;
end Push;

---------------------------------------------------------------------

separate (Stack_Object_Template)

procedure Pop (Element : out Element_Type) is
begin
   if Top = 0 then
      raise Stack_Underflow;
   else
      Element := Element_List (Top);
      Top := Top - 1;
   end if;
end Pop;
```

INSTRUCTOR NOTES

THIS IS A CONTRIVED EXAMPLE.

POINT OUT ONE INSTANTIATION FOR EACH STACK OBJECT.

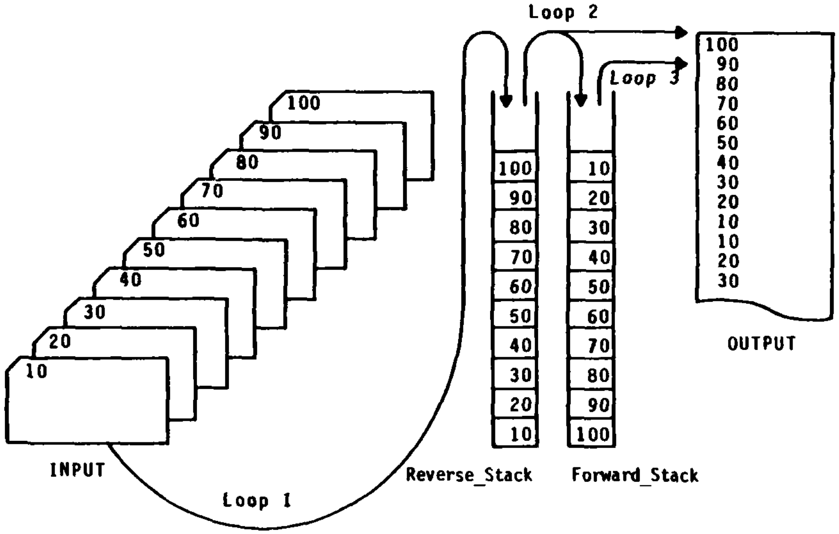
POINT OUT THE $\boxed{\text{OBJECT}}$. $\boxed{\text{OPERATION}}$ NOTATION.

WALK THROUGH THE THREE LOOPS.

SINGLE-OBJECT GENERIC PACKAGES: EXAMPLE OF USE

PROGRAM TO READ A SEQUENCE OF 10 INTEGERS AND PRINT THE SEQUENCE FIRST IN REVERSE ORDER AND THEN FORWARD:

```
with Text_IO; use Text_IO;
procedure Print_Sequence is

   package Type_Integer_IO is new Integer_IO (Integer);
   use Type_Integer_IO;
   package Reverse_Stack is new
      Stack_Object_Template (Element_Type =>  Integer, Capacity =>  10);
   package Forward_Stack is new
      Stack_Object_Template (Element_Type =>  Integer, Capacity =>  10);
   N : Integer;

begin -- Print_Sequence

   for I in 1 .. 10 loop
      Get (N);
      Reverse_Stack.Push (N);
   end loop;
   while not Reverse_Stack.Is_Empty loop
      Reverse_Stack.Pop(N);
      Put (N);
      New_Line;
      Forward_Stack.Push(N);
   end loop;
   while not Forward_Stack.Is_Empty loop
      Forward_Stack.Pop(N);
      Put(N);
      New_Line;
   end loop;

end Print_Sequence;
```



INSTRUCTOR NOTES

SECTION 15

TREES

INSTRUCTOR NOTES

THE TERM HIERARCHY IMPLIES NON-CIRCULARITY. THAT IS, NO NODE IS AN ANCESTOR OF ITSELF.

TREES

- A TREE IS A HIERARCHY OF DATA ITEMS CALLED NODES.
- EACH NODE IS THE PARENT OF ZERO OR MORE OTHER NODES CALLED ITS CHILDREN. EACH CHILD HAS ONE PARENT.
- THERE IS ONE NODE CALLED THE ROOT OF THE TREE, THAT HAS NO PARENT BUT IS (DIRECTLY OR INDIRECTLY) THE ANCESTOR OF EVERY NODE IN THE TREE.
- A NODE WITHOUT ANY CHILDREN IS CALLED A LEAF.

INSTRUCTOR NOTES

POINT OUT THE ROOT, NODE X, AND ITS PARENT AND CHILDREN, AND SEVERAL LEAVES.

A NODE THAT IS NOT A LEAF IS SOMETIMES CALLED AN INTERNAL NODE.

ANSWERS:

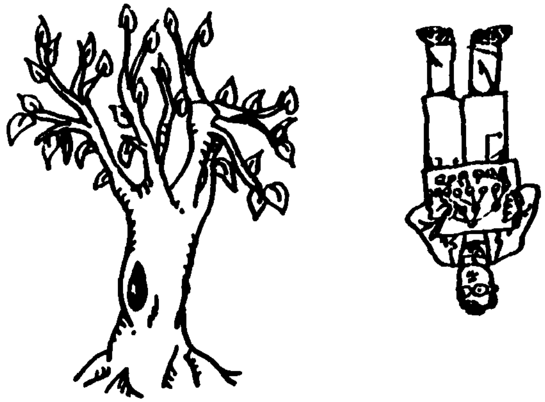
THE PARENT OF NODE 9 IS NODE 3.

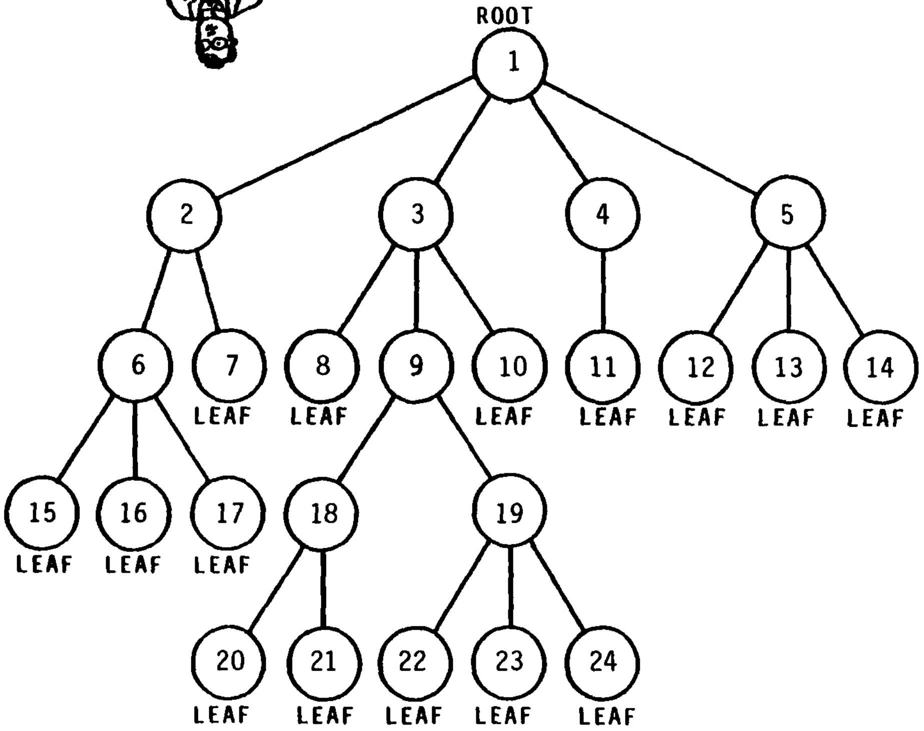
THE CHILDREN OF NODE 9 ARE NODES 18 AND 19.

COMPUTER SCIENTISTS DRAW TREES UPSIDE DOWN.

PARENT OF NODE 9: □

CHILDREN OF NODE 9: □





INSTRUCTOR NOTES

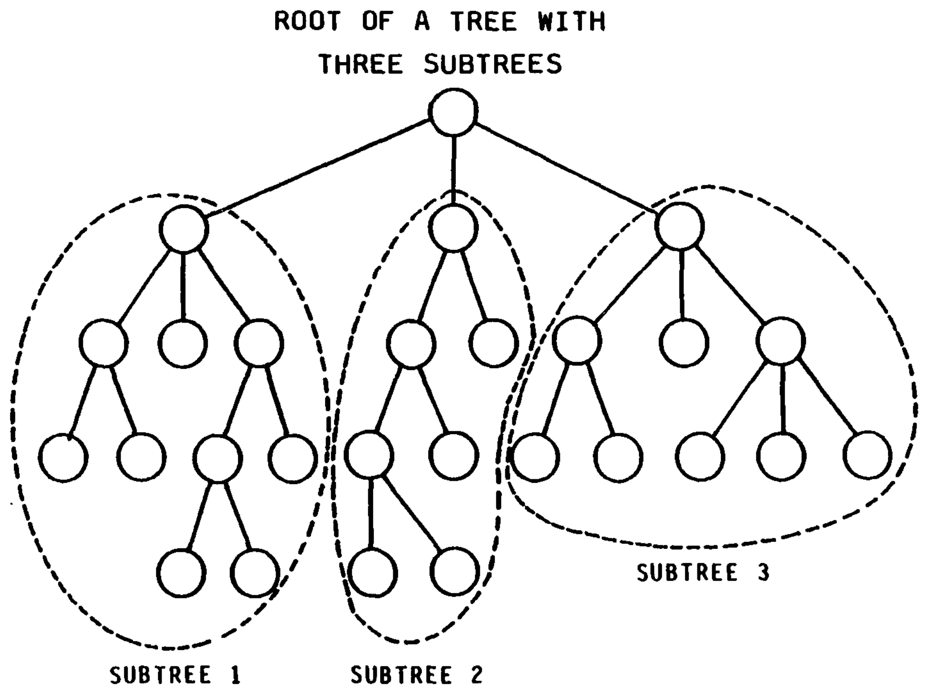
SOMETIMES WE SPEAK INTERCHANGEABLY OF A NODE IN A TREE AND THE SUBTREE OF WHICH THAT NODE IS A ROOT.

SUBTREES

- A CHILD OF THE ROOT MAY ITSELF BE VIEWED AS THE ROOT OF A SMALLER TREE, CONSISTING OF ONE "BRANCH" OF THE ORIGINAL TREE.

- THIS SMALLER TREE IS CALLED A SUBTREE OF THE ORIGINAL TREE.

- EVERY NODE IN A TREE IS THE ROOT OF SOME SUBTREE. (LEAVES ARE ROOTS OF TREES CONSISTING OF A SINGLE NODE.)



INSTRUCTOR NOTES

EXCEPT FOR ONE OR TWO SMALL ILLUSTRATIVE PROGRAMS LATER IN THIS SECTION, ALGORITHMS USING TREES WILL NOT BE PRESENTED UNTIL THE NEXT SECTION.

SO WHAT?

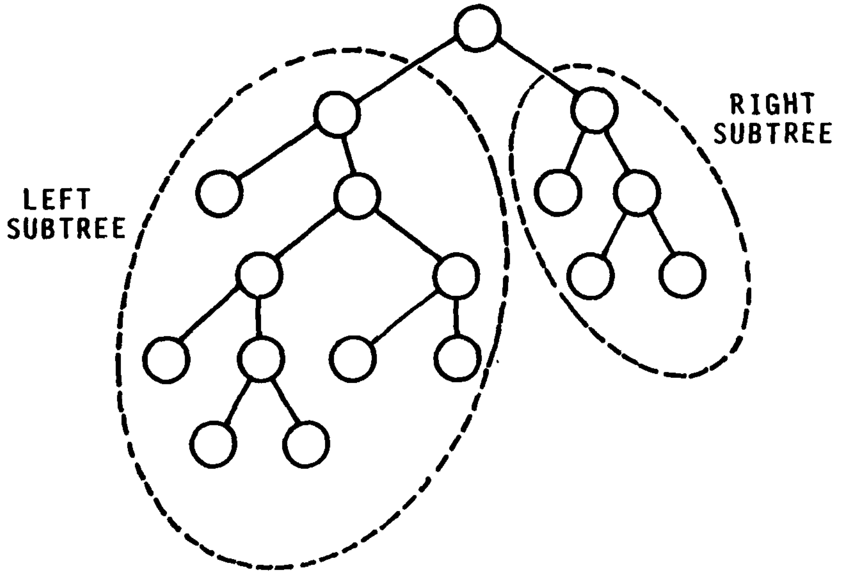
- MANY IMPORTANT ALGORITHMS AND DATA STRUCTURES ARE BUILT AROUND TREES.
- OFTEN THESE ALGORITHMS ARE RECURSIVE. SUBPROGRAMS TO PERFORM SOME OPERATION ON ALL NODES OF A TREE CAN BE CALLED RECURSIVELY WITH EACH OF THE TREE'S SUBTREES, TO PERFORM THE OPERATION ON ALL NODES OF THE SUBTREE.
- FOR NOW, WE'LL CONCENTRATE ON DATA STRUCTURES TO <u>REPRESENT</u> TREES.

INSTRUCTOR NOTES

COMMENT ON THIRD BULLET: A BINARY TREE CONSISTING ONLY OF A ROOT DOES NOT HAVE SUBTREES.

BINARY TREES

- A TREE IN WHICH EACH NON-LEAF HAS TWO CHILDREN IS CALLED A BINARY TREE.
- THE CHILDREN OF A NODE IN A BINARY TREE ARE CALLED THE LEFT CHILD AND THE RIGHT CHILD.
- THE SUBTREES OF A BINARY TREE ARE CALLED THE LEFT SUBTREE AND THE RIGHT SUBTREE.



VG 679.2

INSTRUCTOR NOTES

BULLET 2: IF Data_Type IS ITSELF A RECORD TYPE, WE COULD REPLACE Data_Part BY SEVERAL COMPONENTS CORRESPONDING TO THE COMPONENTS OF A Data_Type RECORD. FOR SIMPLICITY, WE ASSUME THAT THERE IS A SINGLE COMPONENT Data_Part HOLDING ALL THE DATA ASSOCIATED WITH A NODE.

FOR THE ROOT NODE, Parent_Part IS null. OTHERWISE IT POINTS TO THE NODE'S PARENT.

FOR A LEAF NODE, Left_Child_Part AND Right_Child_Part ARE NULL. OTHERWISE, THEY POINT TO THE NODE'S RESPECTIVE CHILDREN.

REPRESENTATIONS OF BINARY TREES

- SUPPOSE THE DATA AT EACH NODE BELONGS TO THE TYPE Data_Type.
- GENERAL CASE:

```
type Node_Type;

type Tree_Type is access Node_Type;

type Node_Type is
   record
      Data_Part                                        : Data_Type;
      Parent_Part, Left_Child_Part, Right_Child_Part : Tree_Type;
   end record;
```

- OFTEN, AN ALGORITHM DOES NOT REQUIRE THE ABILITY TO DETERMINE THE PARENT OF A GIVEN NODE. THEN Node_Type CAN BE SIMPLIFIED:

```
type Node_Type is
   record
      Data_Part                           : Data_Type;
      Left_Child_Part, Right_Child_Part : Tree_Type;
   end record;
```

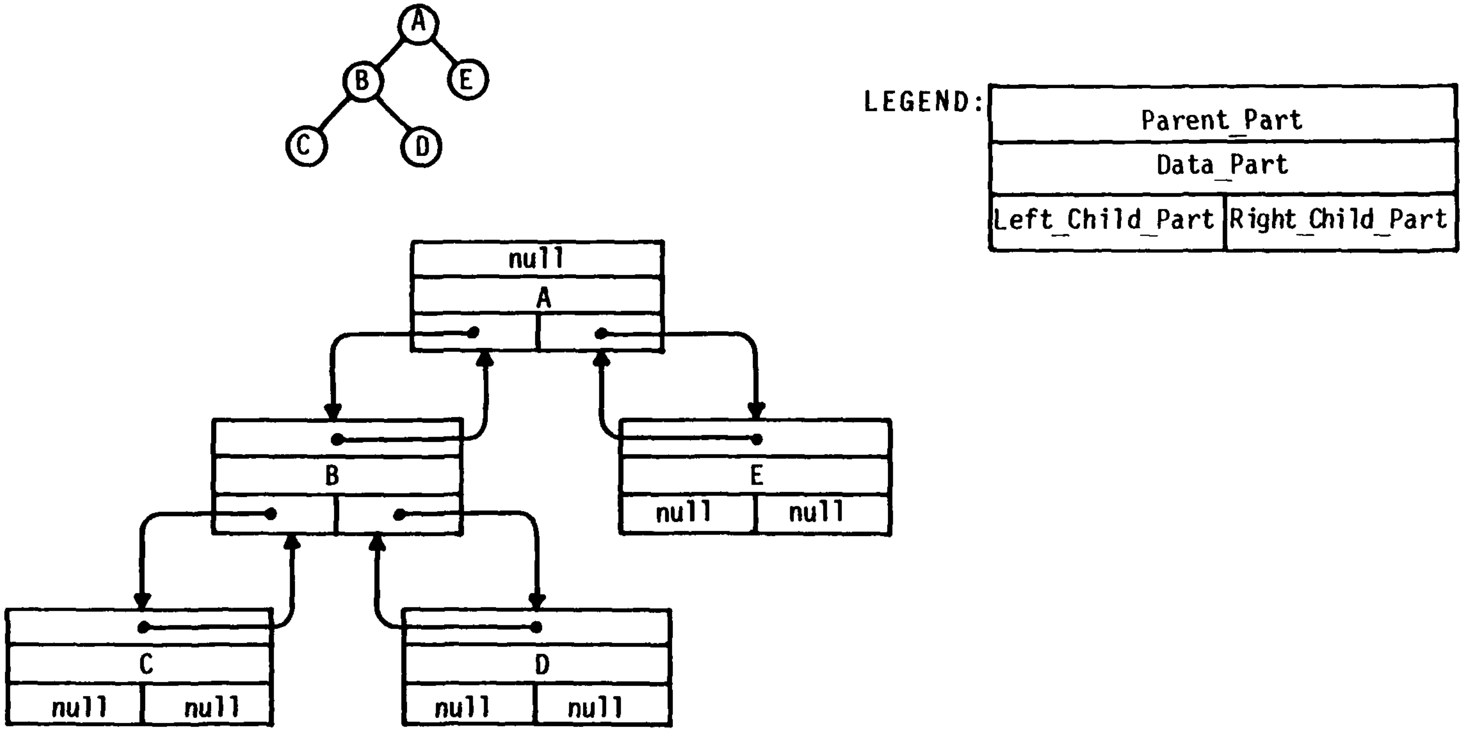
- RARELY, AN ALGORITHM REQUIRES ONLY THE ABILITY TO DETERMINE THE PARENT OF A GIVEN NODE, AND NOT THE CHILDREN. THEN Node_Type CAN BE DECLARED AS FOLLOWS:

```
type Node_Type is
   record
      Data_Part   : Data_Type;
      Parent_Part : Tree_Type;
   end record;
```

INSTRUCTOR NOTES

THIS SLIDE SHOWS AN ABSTRACT VIEW OF A BINARY TREE PLUS ITS INTERNAL REPRESENTATION.

EXAMPLE OF BINARY TREE REPRESENTATION



INSTRUCTOR NOTES

BULLET 2 GIVES A MATHEMATICAL PROPERTY OF BINARY TREES.

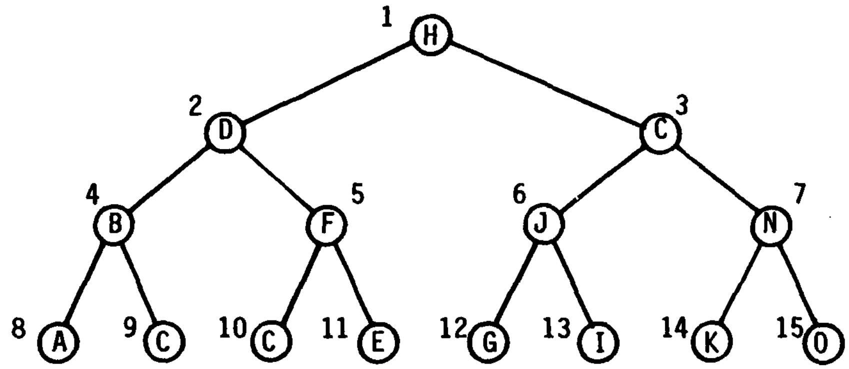
BULLET 3 GIVES A DATA REPRESENTATION EXPLOITING THIS PROPERTY.

ADDENDUM TO BULLET 4: BECAUSE INTEGER DIVISION TRUNCATES, NOTE NUMBER I/2 IS ALWAYS THE PARENT OF NODE NUMBER I FOR I GREATER THAN 1 (WHETHER NODE NUMBER J IS A LEFT SON OR RIGHT SON).

THIS REPRESENTATION DOES NOT ALLOW FOR THE ADDITION OR DELETION OF NODES.

A CONVENIENT REPRESENTATION FOR BINARY TREES WITH UNIFORM DEPTH

- NUMBER THE NODES LEVEL BY LEVEL, STARTING AT THE TOP, GOING LEFT-TO-RIGHT IN EACH ROW. BEGIN BY ASSIGNING THE NUMBER 1 TO THE ROOT.



- IF A NODE IS NUMBERED n, ITS LEFT CHILD IS NUMBERED 2n AND ITS RIGHT CHILD IS NUMBERED 2n + 1.

- STORE THE INTERNAL DATA OF THE NODES IN AN ARRAY INDEXED BY NODE NUMBERS.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	D	C	B	F	J	N	A	C	C	E	G	I	K	O

- THERE IS NO NEED TO STORE LINK INFORMATION. IF Node_Data (I) CONTAINS THE INTERNAL DATA FOR SOME NODE, Node_Data (2 * I) AND Node_Data (2 * I + 1) CONTAIN THE INTERNAL DATA FOR ITS LEFT CHILD AND RIGHT CHILD, RESPECTIVELY.

INSTRUCTOR NOTES

DATA COMPRESSION IS USEFUL FOR ARCHIVING OLD DATA. THE NECESSITY TO "UNCOMPRESS" THE DATA MAKES IT INCONVENIENT TO ACCESS THE DATA ON A REGULAR BASIS, BUT THE ARCHIVES REQUIRE LESS SPACE.

BULLET 2: THE FORMULATION OF A GOOD HUFFMAN CODE ASSUMES A CONTEXT IN WHICH CHARACTERS OCCUR WITH PREDICTABLE FREQUENCIES, E.G., ENGLISH TEXT.

BULLET 3: THE SMALLEST UNIFORM LENGTH CHARACTER CODE FOR 26 LETTERS WOULD REQUIRE 5 BITS PER CHARACTER. THE WORD "ENCODING" WOULD REQUIRE 40 BITS IN SUCH A CODE. THE HUFFMAN CODE SHOWN USES ONLY 33 BITS, 17.5% LESS.

BULLET 4: POINT OUT, FOR EXAMPLE, THAT THE CODE FOR E IS 111 AND NO OTHER CODE BEGINS WITH THESE BITS. THUS WE ALWAYS KNOW TO INTERPRET 111 AS A COMPLETE CHARACTER. THE SAME PROPERTY APPLIES TO ALL OTHER LETTERS.

ANSWER TO QUIZ: $\underbrace{1\ 1\ 0\ 1}_{A}\ \underbrace{1\ 0\ 0\ 0\ 1}_{D}\ \underbrace{1\ 1\ 0\ 1}_{A}$

AN APPLICATION FOR BINARY TREES: HUFFMAN CODES

- HUFFMAN CODES ARE A MEANS FOR COMPRESSING DATA -- REPRESENTING THE SAME AMOUNT OF INFORMATION IN FEWER BITS.
- INSTEAD OF USING THE SAME NUMBER OF BITS FOR EACH CHARACTER CODE, WE ASSIGN SHORT BIT REPRESENTATIONS TO FREQUENTLY OCCURRING CHARACTERS, LONG BIT REPRESENTATIONS TO RARELY OCCURRING CHARACTERS.
- A TYPICAL HUFFMAN CODE FOR TEXT CONSISTING ENTIRELY OF CAPITAL LETTERS:

A	1101	N	1011
B	1000000	O	010
C	00101	P	00011
D	10001	Q	00000001
E	111	R	1010
F	00001	S	0011
G	001000	T	1001
H	11001	U	00010
I	011	V	1000001
J	000000001	W	000001
K	00000010	X	00000011
L	11000	Y	001001
M	100001	Z	000000000

$$\underbrace{1\,1\,1}_{E}\;\underbrace{1\,0\,1\,1}_{N}\;\underbrace{0\,0\,1\,0\,1}_{C}\;\underbrace{0\,1\,0}_{O}\;\underbrace{1\,0\,0\,0\,1}_{D}\;\underbrace{0\,1\,1}_{I}\;\underbrace{1\,0\,1\,1}_{N}\;\underbrace{0\,0\,1\,0\,0\,0}_{G}$$

- HOW DO WE KNOW WE HAVE REACHED THE END OF ONE CHARACTER'S CODE?
  - -- HUFFMAN CODES ARE CONSTRUCTED SO THAT NO LETTER'S BIT SEQUENCE BEGINS WITH THE COMPLETE BIT SEQUENCE FOR ANOTHER LETTER.
  - -- AS SOON AS WE'VE SEEN A SEQUENCE OF BITS THAT CAN BE INTERPRETED AS REPRESENTING A LETTER, WE SHOULD INTERPRET IT THAT WAY.
- QUIZ: WHAT IS THE CODE FOR "ADA"?

INSTRUCTOR NOTES

THE TREE SHOWN IS ANOTHER REPRESENTATION FOR THE CODE GIVEN ON THE PREVIOUS SLIDE.

BULLET 1: EACH LEAF CORRESPONDS TO ONE OF THE ENCODED CHARACTERS. TO FIND A CHARACTER'S CODE, FOLLOW THE PATH FROM THE ROOT TO THE CORRESPONDING LEAF. BUILD THE CODE LEFT-TO-RIGHT BY APPENDING A 0 WHEN TAKING A LEFT BRANCH AND APPENDING A 1 WHEN TAKING A RIGHT BRANCH. THE CODE FOR N IS 1011.

(THE NOTATIONS s0 AND s1 ON THE SLIDE MEAN THE BITS IN SEQUENCE s FOLLOWED BY THE BIT 0 OR THE BIT 1, RESPECTIVELY.)

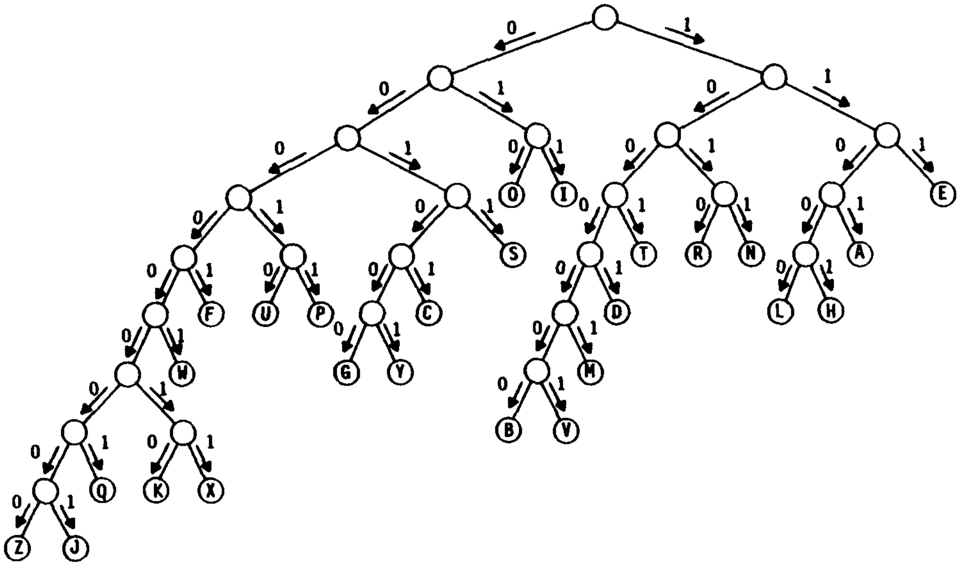
BULLET 2: THIS IS AN INTUITIVE PROOF THAT A HUFFMAN CODE HAS THE PROPERTY REQUIRED TO ALLOW US TO RECOGNIZE THE END OF A CHARACTER'S CODE.

BULLET 3: TO DECODE A SEQUENCE OF BITS, START AT THE ROOT AND GO TO THE LEFT CHILD WHEN READING A ZERO, TO THE RIGHT CHILD WHEN READING A ONE. WHEN A LEAF IS REACHED, YOU HAVE READ THE CODE FOR THE CORRESPONDING CHARACTER. OUTPUT THE CHARACTER AND GO BACK TO THE ROOT FOR THE NEXT BIT.

THE ALGORITHM WILL BE GIVEN SHORTLY IN ADA.

WALK STUDENTS THROUGH THE FIRST ENCODED LETTER OF THE QUIZ AND HAVE THEM COMPLETE THE WORD, ANSWER: "TREES"

HUFFMAN CODES CAN BE REPRESENTED AS BINARY TREES



- EACH NODE AT LEVEL n OF THE TREE (COUNTING THE TOP LEVEL AS LEVEL 0) CORRESPONDS TO A SEQUENCE OF n BITS.

  -- THE ROOT CORRESPONDS TO THE EMPTY SEQUENCE.
  -- IF A NODE CORRESPONDS TO THE SEQUENCE s, ITS LEFT CHILD CORRESPONDS TO THE SEQUENCE s0 AND ITS RIGHT SEQUENCE CORRESPONDS TO THE SEQUENCE s1.
  -- LEAVES CORRESPOND TO COMPLETE BIT SEQUENCES FOR LETTERS

- BECAUSE THE NODES CORRESPONDING TO LETTERS HAVE NO CHILDREN, ONE LETTER'S BIT SEQUENCE CANNOT BEGIN WITH ANOTHER LETTER'S COMPLETE BIT SEQUENCE.

- THIS TREE CAN BE USED FOR DECODING HUFFMAN-CODED MESSAGES.

- QUIZ: DECODE 1001101011111110011

INSTRUCTOR NOTES

THE REPLACEMENT FOR Data_Part IN THE LEAF VARIANT IS Letter_Part.

THE REPLACEMENT FOR Data_Part IN THE NON-LEAF VARIANT IS EMPTY.

TREES IN WHICH LEAVES AND NON-LEAVES CONTAIN DIFFERENT INFORMATION

- IN MANY APPLICATIONS, LEAVES AND OTHER NODES CONTAIN DIFFERENT KINDS OF DATA.

- HUFFMAN TREES ARE AN EXAMPLE.
  - -- LEAVES CONTAIN A CHARACTER GIVING THE CORRESPONDING LETTER
  - -- OTHER NODES CONTAIN NO DATA

- Node_Type SHOULD THEN BE DEFINED AS A TYPE WITH VARIANTS.
  - -- EACH VARIANT MAY CONTAIN ITS OWN REPLACEMENT FOR Data_Part.
  - -- THE Left_Child_Part AND Right_Child_Part COMPONENTS NEED ONLY BE PRESENT IN THE NON-LEAF VARIANT.

- HUFFMAN EXAMPLE:

```
type Node_Type (Is_A_Leaf : Boolean);

type Tree_Type is access Node_Type;

type Node_Type (Is_A_Leaf : Boolean) is
   record
      case Is_A_Leaf is
         when False =>
            Left_Child_Part, Right_Child_Part : Tree_Type;
         when True =>
            Letter_Part : Character range 'A' .. 'Z';
      end case;
   end record;
```

INSTRUCTOR NOTES

Get_Letter IS TO PERFORM THE COMPUTATION THAT STUDENTS DID BY HAND TWO SLIDES EARLIER, IN TERMS OF THE TYPE DECLARATIONS ON THE PREVIOUS SLIDE.

USE OF HUFFMAN TREES FOR DECODING

- ASSUME WE HAVE ALREADY WRITTEN THE SUBPROGRAM

```
procedure Get_Bit (Bit : out Boolean);
```

TO OBTAIN THE NEXT BIT IN A HUFFMAN-ENCODED BIT STREAM

- WE WANT TO WRITE THE FOLLOWING SUBPROGRAM:

```
procedure Get_Letter (Tree : in Tree_Type; Letter : out Character);
```

(Tree_Type IS AS DEFINED ON THE PREVIOUS SLIDE.) Get_Letter IS TO OBTAIN BITS BY CALLING Get_Bit, DECODE THEM BY USING THE HUFFMAN TREE IN PARAMETER Tree, AND PLACE THE DECODED LETTER IN PARAMETER Letter.

INSTRUCTOR NOTES

SINCE Tree_Type IS AN ACCESS TYPE, Subtree.Is_A_Leaf, Subtree.Right_Child_Part, Subtree.Left_Child_Part, and Subtree.Letter_Part NAME COMPONENTS OF THE Node_Type RECORD DESIGNATED BY Subtree.

THE STATEMENTS INSIDE THE LOOP ARE REACHED ONLY WHEN THE DISCRIMINANT Is_A_Leaf IS FALSE, SO THE Left_Child_Part AND Right_Child_Part COMPONENTS EXIST. THE STATEMENT FOLLOWING THE LOOP IS REACHED ONLY WHEN THE DISCRIMINANT Is_A_Leaf IS True, SO THE Letter_Part COMPONENT EXISTS.

IF NECESSARY, TRACE THROUGH AN INVOCATION OF Get_Letter, USING THE TREE DEPICTED THREE SLIDES EARLIER.

HUFFMAN-DECODING SUBPROGRAM

PULL 4-34 AND 4-35 FROM VG 679

```
with Get_Bit;

procedure Get_Letter (Tree : in Tree_Type; Letter : out Character) is

   Subtree  : Tree_Type := Tree;
   Go_Right : Boolean;

begin -- Get_Letter

   while not Subtree.Is_A_Leaf loop
      Get_Bit (Go_Right);
      if Go_Right then
         Subtree := Subtree.Right_Child_Part;
      else
         Subtree := Subtree.Left_Child_Part;
      end if;
   end loop;

   Letter := Subtree.Letter_Part;

end Get_Letter;
```

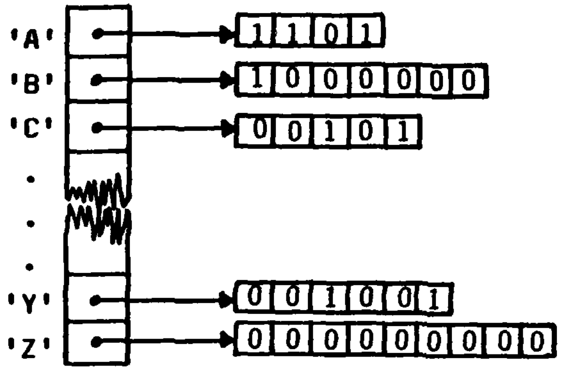
INSTRUCTOR NOTES

IF Letter_Code_Table IS A Letter_Code_Table_Type VARIABLE AND Letter IS A VARIABLE IN Character RANGE 'A' .. 'Z', THEN Letter_Code_Table (Letter).all IS THE HUFFMAN CODE CORRESPONDING TO LETTER.

ENCODING LETTERS INTO HUFFMAN CODES

- THE HUFFMAN TREE IS NOT AS CONVENIENT FOR GOING IN THE OPPOSITE DIRECTION -- TRANSLATING A SEQUENCE OF CHARACTERS INTO A SEQUENCE OF HUFFMAN-ENCODED BITS.

- FOR THAT PURPOSE, A DATA STRUCTURE LIKE THE FOLLOWING WOULD BE MORE CONVENIENT:



(WE USE AN ARRAY OF POINTERS TO BIT SEQUENCES RATHER THAN AN ARRAY OF BIT SEQUENCES BECAUSE THE SEQUENCES HAVE DIFFERENT LENGTHS.)

- ADA TYPE DECLARATIONS:

```
type Bit_Sequence_Type is array (Positive range < >) of Boolean;

type Bit_Sequence_Pointer_Type is access Bit_Sequence_Type;

type Letter_Code_Table_Type is
   array (Character range 'A' .. 'Z') of Bit_Sequence_Pointer_Type;
```

INSTRUCTOR NOTES

RELATE THE THIRD BULLET TO THE FIRST BULLET.

THE ACTUAL CODE FOLLOWS ON THE NEXT SLIDE.

BUILDING A LETTER CODE TABLE FROM A HUFFMAN TREE

- WE WRITE A SUBPROGRAM TAKING THREE PARAMETERS:

  -- A PARTIALLY FILLED-IN LETTER CODE TABLE
  -- A SUBTREE OF THE HUFFMAN TREE
  -- THE BIT SEQUENCE CORRESPONDING TO THE ROOT OF THE SUBTREE

  THE SUBPROGRAM FILLS IN THOSE TABLE ENTRIES CORRESPONDING TO THE LEAVES IN THAT SUBTREE.

- THERE ARE TWO CASES TO CONSIDER:

  -- <u>WHEN THE SUBTREE CONSISTS OF A SINGLE LEAF</u>:

  THE THIRD PARAMETER IS THE BIT SEQUENCE CORRESPONDING TO THE LEAF. ENTER THIS BIT SEQUENCE GIVEN IN THE TABLE AT THE POSITION INDICATED BY THE LETTER IN THE LEAF.

  -- <u>WHEN THE SUBTREE ITSELF HAS LEFT AND RIGHT SUBTREES</u>:

  CALL THE SUBPROGRAM RECURSIVELY WITH THE LEFT AND RIGHT SUBTREES TO FILL IN THE TABLE ENTRIES CORRESPONDING TO THE LEAVES OF EACH SUBTREE.

  IF THE BIT SEQUENCE GIVEN BY THE THIRD PARAMETER IS X, THE BIT SEQUENCES CORRESPONDING TO THE ROOTS OF THE LEFT AND RIGHT SUBTREES ARE X & *False* AND X & *True*, RESPECTIVELY.

- THE BIT SEQUENCE CORRESPONDING TO THE ROOT OF THE ENTIRE HUFFMAN TREE IS THE EMPTY SEQUENCE, SO A CALL WITH THE FOLLOWING PARAMETERS FILLS IN THE ENTIRE TABLE:

  -- AN INITIALLY EMPTY LETTER CODE TABLE (THAT WILL BE COMPLETELY FILLED UPON THE RETURN)

  -- THE ENTIRE HUFFMAN TREE

  -- THE EMPTY BIT SEQUENCE

INSTRUCTOR NOTES

THIS IS THE ADA IMPLEMENTATION OF THE ALGORITHM ON THE PREVIOUS SLIDE. MANY TREE MANIPULATION ALGORITHMS HAVE THE SAME BASIC FORM.

THE TYPES Letter_Code_Table_Type, Tree_Type, AND Bit_Sequence_Type WERE GIVEN ON PREVIOUS SLIDES.

A TYPICAL RECURSIVE TREE-MANIPULATION SUBPROGRAM

```
procedure Fill_In_Table
   (Letter_Code_Table : in out Letter_Code_Table_Type;
    Subtree           : in Tree_Type;
    Root_Bit_Sequence : in Bit_Sequence_Type) is

begin

   if Subtree.Is_A_Leaf then
      Letter_Code_Table (Subtree.Letter_Part) :=
         new Bit_Sequence_Type'(Root_Bit_Sequence);
   else
      Fill_In_Table
         (Letter_Code_Table,
          Subtree.Left_Child_Part,
          Root_Bit_Sequence & False);
      Fill_In_Table
         (Letter_Code_Table,
          Subtree.Right_Child_Part,
          Root_Bit_Sequence & True);
   end if;

end Fill_In_Table;
```

INSTRUCTOR NOTES

SINCE THE INPUT VALUE OF THE FIRST PARAMETER TO Fill_In_Table IS IRRELEVANT ON THE INITIAL CALL AND THE VALUE OF THE THIRD PARAMETER IS REQUIRED TO HAVE A PARTICULAR FIXED VALUE, WE CAN PROVIDE THE OUTSIDE USER OF Fill_In_Table WITH A SIMPLIFIED INTERFACE MORE APPROPRIATE TO HIS LEVEL OF ABSTRACTION.

THE INTERFACE IS A FUNCTION TAKING A HUFFMAN TREE AS A PARAMETER AND RETURNING THE CORRESPONDING TABLE. THIS FUNCTION IS IMPLEMENTED USING A SINGLE CALL ON Fill_In_Table.

POINT OUT THE NULL BIT SEQUENCE.

INSTEAD OF NESTING Fill_In_Table INSIDE Table_From_Tree, WE COULD HAVE WRITTEN A PACKAGE WHOSE DECLARATION DECLARES ONLY Table_From_Tree AND WHOSE BODY CONTAINS BOTH THE Fill_In_Table PROCEDURE BODY AND THE Table_From_Tree FUNCTION BODY. OPINIONS VARY ABOUT WHICH STYLE IS PREFERABLE. (ONE ADVANTAGE OF NESTING IN THIS CASE IS THAT THE "TOP-LEVEL" CALL ON Fill_In_Table IS PHYSICALLY CLOSE TO THE PROCEDURE BODY AND HELPS A READER UNDERSTAND THE PROCEDURE BODY.)

SIMPLIFYING THE INTERFACE

```
function Table_From_Tree (Tree : Tree_Type) return Letter_Code_Table_Type is

   Empty_Sequence : Bit_Sequence_Type (1 .. 0);
   Result         : Letter_Code_Table_Type;

   procedure Fill_In_Table (...; ...; ...) is
      ...
   end Fill_In_Table;

begin -- Table_From_Tree

   Fill_In_Table (Result, Tree, Empty_Sequence);
   return Result;

end Table_From_Tree;
```

INSTRUCTOR NOTES

ADDING CHILDREN TO A LINKED LIST IS ESPECIALLY EASY IF THE ORDER OF THE CHILDREN IS IRRELEVANT, BECAUSE INSERTION AT THE FRONT OF A LINKED LIST IS EASY.

A LINKED LIST OF CHILDREN MAY BE SINGLY- OR DOUBLY-LINKED. A DOUBLY-LINKED LIST MAKES DELETION OF CHILDREN EASIER, BUT THE REST OF THIS SECTION USES A SINGLY-LINKED LIST FOR SIMPLICITY.

NON-BINARY TREES

- ASSOCIATED WITH EACH NODE IS A <u>LIST</u> OF CHILDREN.
- THE LENGTH OF THE LIST IS ARBITRARY.
- EITHER A LINEAR LIST OR A LINKED LIST MAY BE USED.
  - -- A LINEAR LIST MAKES IT EASY TO FIND THE $\underline{n}^{th}$ SUBTREE.
  - -- A LINKED LIST MAKES IT EASY TO ADD OR REMOVE CHILDREN OF AN EXISTING NODE.

INSTRUCTOR NOTES

THE LIST OF CHILDREN IS REPRESENTED BY AN ARRAY. SINCE THE SIZE OF THE ARRAY WILL VARY FROM NODE TO NODE, ITS SIZE IS GIVEN BY THE DISCRIMINANT Number_Of_Children. NOTE THAT THE DISCRIMINANT APPEARS IN BOTH THE INCOMPLETE TYPE DECLARATION AND THE FULL TYPE DECLARATION.

THE DISCRIMINANT EQUALS 0 IN LEAF NODES, MAKING Child_List_Part A NULL ARRAY.

THE THIRD COMPONENT OF Node_Type CAN BE VIEWED EQUIVALENTLY AS A LIST SUBTREES OR A LIST OF CHILDREN. THE CHILDREN ARE THE NODES AT THE ROOT OF THE SUBTREES.

SINCE A DISCRIMINANT PERMANENTLY CONSTRAINS AN ALLOCATED VARIABLE, THE ONLY WAY TO ADD OR DELETE CHILDREN IS TO ALLOCATE A NEW Node_Type VARIABLE TO REPLACE THE OLD ONE.

AGAIN, Parent_Part MAY BE ABSENT FOR MANY APPLICATIONS. IN THE RARE CASE THAT Child_List_Part IS ABSENT, THE DISCRIMINANT MAY ALSO BE REMOVED, AND THE RESULTING DATA TYPE IS THE SAME AS FOR BINARY UPWARD-POINTING-ONLY TREES.

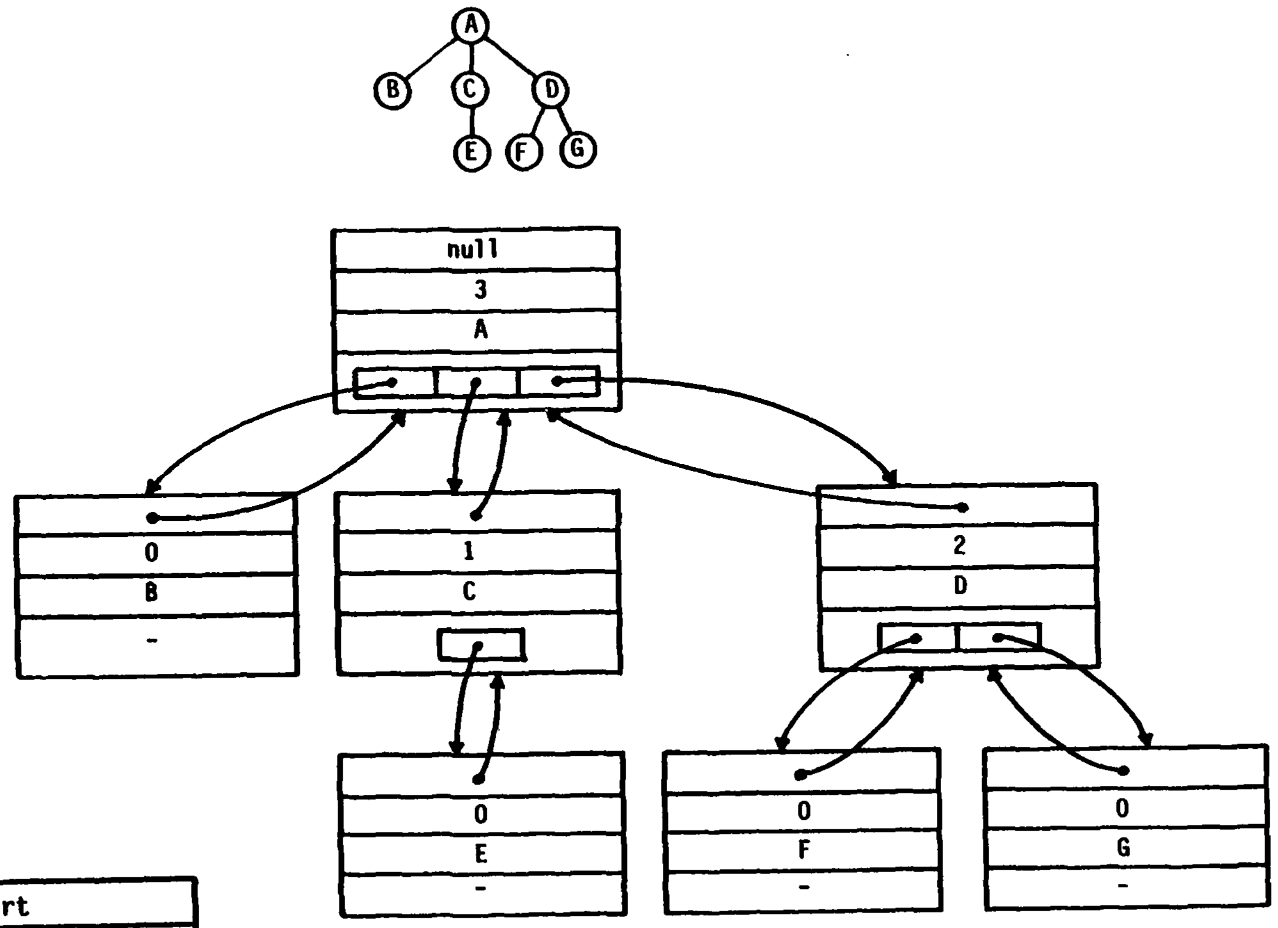
LINEAR LIST REPRESENTATION

```
type Node_Type (Number_Of_Children : Natural);

type Tree_Type is access Node_Type;

type Subtree_List_Type is array (Positive range < >) of Tree_Type;

type Node_Type (Number_Of_Children : Natural) is
   record
      Data_Part       : Data_Type;
      Parent_Part     : Tree_Type;
      Child_List_Part : Subtree_List_Type (1 .. Number_Of_Children);
   end record;
```

INSTRUCTOR NOTES

NODE A HAS A 3-ELEMENT ARRAY, NODE D HAS A 2-ELEMENT ARRAY, NODE C HAS A 1-ELEMENT ARRAY, AND THE REMAINING NODES HAVE 0-ELEMENT ARRAYS.

EXAMPLE OF LINEAR-LIST REPRESENTATION



LEGEND:

Parent_Part
Number_Of_Children
Data_Part
Child_List_Part

(- = null array)

INSTRUCTOR NOTES

THE DISCRIMINANT NOW CONTROLS BOTH A VARIANT PART AND THE INDEX CONSTRAINT IN THE SECOND VARIANT. THE Parent_Part COMPONENT IS PRESENT IN ALL NODES.

LINEAR LIST REPRESENTATION FOR DIFFERENT DATA AT LEAVES AND NON-LEAVES

- ASSUMES LEAVES HAVE DATA OF TYPE Leaf_Data_Type, OTHER NODES HAVE DATA OF TYPE Non_Leaf_Data_Type.

- REVISED DECLARATION OF Node_Type:

```
type Node_Type (Number_Of_Children : Natural) is
   record
      Parent_Part : Tree_Type;
      case Number_Of_Children is
         when 0 =>
           Leaf_Data_Part : Leaf_Data_Type;
         when others =>
           Non_Leaf_Data_Part : Non_Leaf_Data_Type;
           Child_List_Part    : Subtree_List_Type (1 .. Number_Of_Children);
      end case;
   end record;
```

INSTRUCTOR NOTES

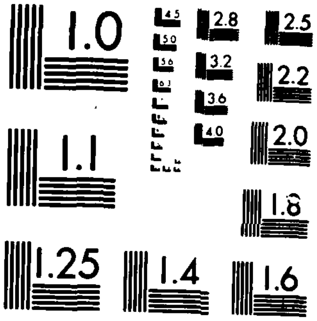
TO REPRESENT CHILD LISTS AS LINKED LISTS, WE INSTANTIATE THE GENERIC LINKED LIST PACKAGE DEVELOPED EARLIER.

THE LIST ELEMENTS ARE SUBTREES, OF TYPE Tree_Type.

THERE IS NO LONGER ANY NEED FOR A DISCRIMINANT.

AD-A165 076 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA 5/7
TOPICS L305 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC
WALTHAM MA 1986 DAAB07-83-C-K506

UNCLASSIFIED F/G 9/2 NL



1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

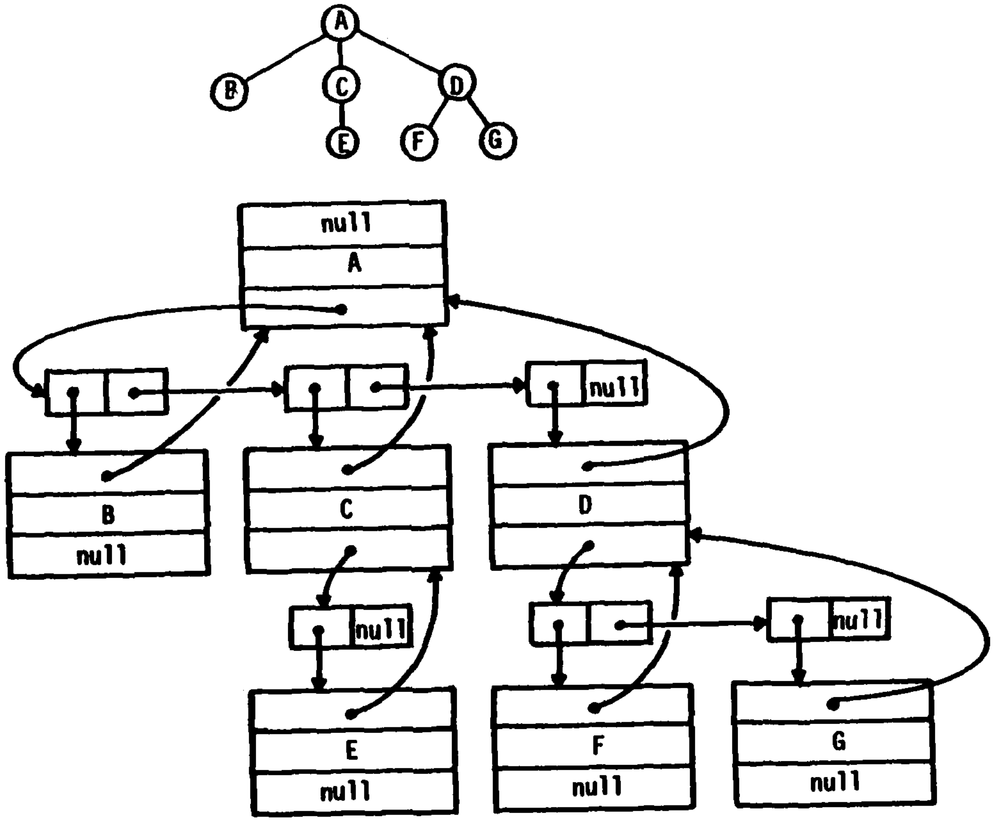
LINKED-LIST REPRESENTATION

```
type Node_Type;

type Tree_Type is access Node_Type;

package Subtree_List_Package is
   new Linked_List_Package_Template (Element_Type =>  Tree_Type);

type Node_Type is
   record
      Data_Part       : Data_Type;
      Parent_Part     : Tree_Type;
      Child_List_Part : Subtree_List_Package.List_Type;
   end record;
```

INSTRUCTOR NOTES

POINT OUT THE LINKED LISTS. OBSERVE THAT THERE IS EXACTLY ONE LIST CELL FOR EACH TREE NODE OTHER THAN THE ROOT. (THE NEXT TWO SLIDES EXPLOIT THIS OBSERVATION TO DEVELOP A MORE EFFICIENT REPRESENTATION.)

EXAMPLE OF LINKED-LIST REPRESENTATION



LEGEND:

TREE NODES:

Parent_Part
Data_Part
Child_List_Part

LIST CELLS:

List Element	Forward Link

(LINKED LISTS ARE IMPLEMENTED WITHOUT DUMMY CELLS)

INSTRUCTOR NOTES

Next_Sibling_Part POINTS TO THE NEXT CHILD OF A NODE'S PARENT. Child_List_Part POINTS TO A NODE'S OWN FIRST CHILD.

THIS REPRESENTATION PREVENTS A SINGLE SUBTREE FROM BEING SHARED BY SEVERAL TREES SIMULTANEOUSLY. SINCE THE STUDENTS WERE PROBABLY NOT AWARE OF THIS POSSIBILITY, IT IS BEST NOT TO BRING THIS UP IF THEY DON'T BRING IT UP FIRST.

AN IMPROVED LINKED-LIST REPRESENTATION

- <u>OBSERVATION</u>: THERE IS EXACTLY ONE LINKED LIST CELL CORRESPONDING TO EACH TREE NODE (OTHER THAN THE ROOT).

- <u>CONCLUSION</u>: THE LIST CELLS AND CORRESPONDING TREE NODES CAN BE MERGED, SO THAT EACH TREE NODE CONTAINS A POINTER TO THE NEXT TREE NODE IN THE LIST OF CHILDREN.

```
type Node_Type;

type Tree_Type is access Node_Type;

type Node_Type is
   record
      Data_Part                                          : Data_Type;
      Parent_Part, Next_Sibling_Part, Child_List_Part : Tree_Type;
   end record;
```

- <u>BENEFITS</u>: WE SAVE TIME AND SPACE

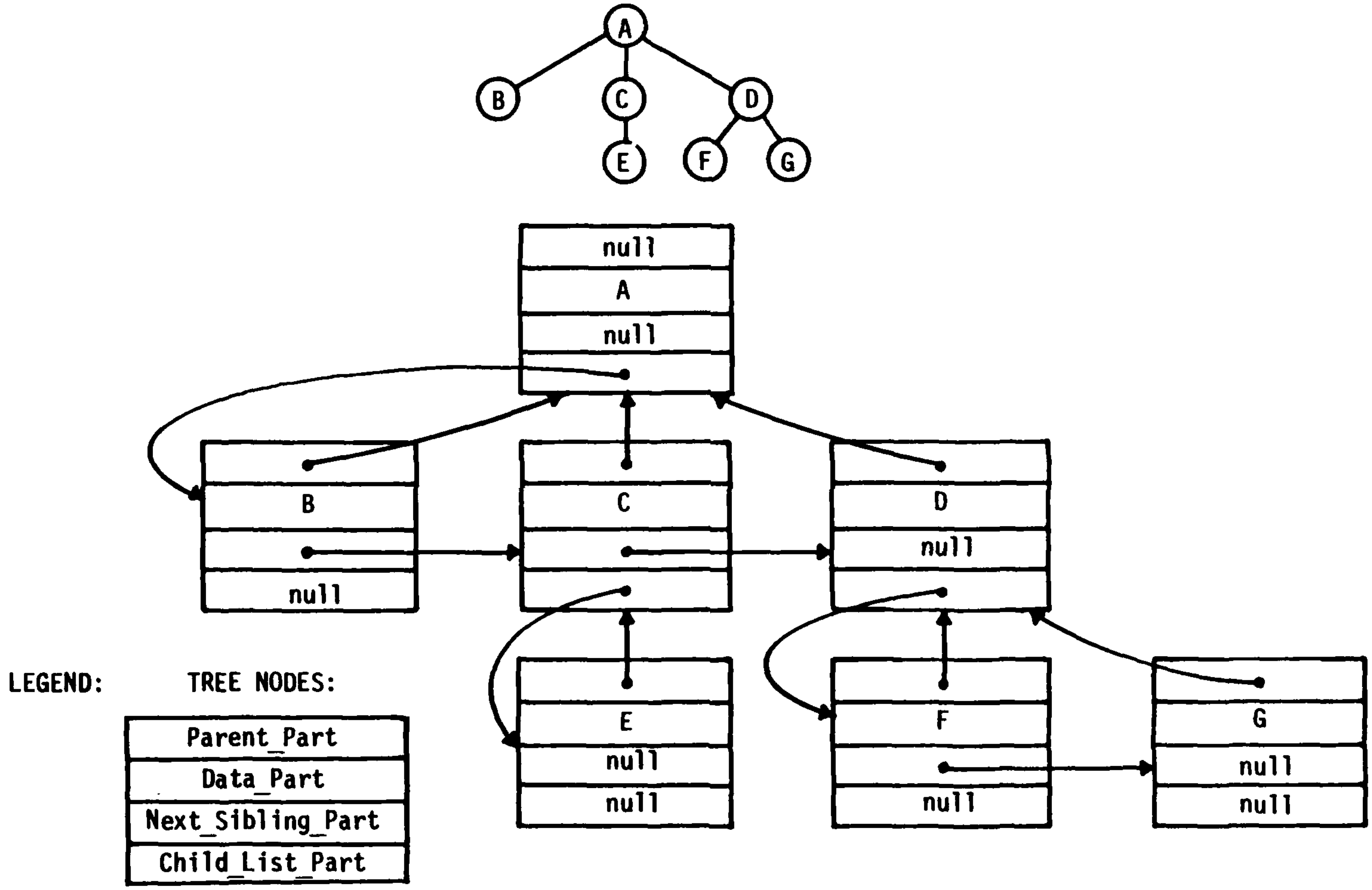
  -- THE SPACE TAKEN UP BY THE NODE POINTERS IN THE LIST CELLS

  -- THE TIME TAKEN TO FOLLOW THOSE POINTERS

INSTRUCTOR NOTES

TRACE THROUGH THE LINKED LIST OF A'S CHILDREN.

EXAMPLE OF IMPROVED LINKED-LIST REPRESENTATION



INSTRUCTOR NOTES

SECTION 16

SEARCHING

INSTRUCTOR NOTES

SEARCHING IS SOMETHING THAT OCCURS FREQUENTLY, SO WE LOOK AT SEVERAL SEARCHING ALGORITHMS.

SEARCHING REQUIREMENTS VARY GREATLY:

- NUMBER OF ELEMENTS
- NUMBER OF SEARCHES
- MEMORY SIZE
- PERFORMANCE

PICKING AN ALGORITHM REQUIRES KNOWING SOMETHING ABOUT ITS PERFORMANCE SO WE WILL LOOK AT HOW TO COMPARE ALGORITHMS BY PERFORMANCE.

SEARCHING -- TOPICS

- PERFORMANCE
- LINEAR SEARCH
- BINARY SEARCH
- SEARCH TREES
- HASHING
- PRIORITY QUEUES

FURTHER DETAILS IN EXERCISE 5.1 OF THE ADVANCED Ada WORKBOOK.

INSTRUCTOR NOTES

A GROSS PERFORMANCE MEASURE OF AN ALGORITHM CAN BE GIVEN IN TERMS OF ITS DOMINANT OPERATION, THE COMPARE OPERATION IN SEARCHING. WE USE ORDER NOTATION TO INDICATE HOW THE AVERAGE PERFORMANCE BEHAVES.

FOR EXAMPLE, IF A SEARCH ALGORITHM IS ORDER (n) THEN THE ACTUAL NUMBER OF COMPARISONS IS SOME MULTIPLE OF n PLUS A CONSTANT, E.G. $2n+1$, $1/2n$, ETC.

THIS DIVIDES ALGORITHMS FOR A PROBLEM INTO CLASSES. ANY TWO, SAY ORDER (n), ALGORITHMS HAVE, IN A GROSS SENSE, THE SAME PERFORMANCE. THE ONE TO PICK MIGHT DEPEND ON THE EXACT NUMBER OF COMPARISONS, THE MEMORY AVAILABLE, OR THE AVAILABILITY OF A SEARCH PACKAGE.

PERFORMANCE OF AN ALGORITHM

- USES DOMINANT OPERATION, E.G. THE <u>COMPARE</u> OPERATION IN SEARCHING

- USES <u>ORDER</u> NOTATION TO EXPRESS AVERAGE PERFORMANCE

  EXAMPLE:

  LET n BE THE NUMBER OF ITEMS TO BE SEARCHED.

  A SEARCH ALGORITHM IS ORDER (n) IF

  $$\frac{\text{NUMBER OF COMPARISONS}}{n} \approx \text{CONSTANT}$$

  FOR LARGE n

- THE ALGORITHMS TO SOLVE A PROBLEM MAY VARY GREATLY IN PERFORMANCE:

ORDER (n^2)	⇓	BAD PERFORMANCE
ORDER ($n\log_2 n$)		
ORDER (n)		
ORDER ($\log_2 n$)		GOOD PERFORMANCE

INSTRUCTOR NOTES

THERE ARE THEORETICAL PROOFS THAT A SORTING ALGORITHM CANNOT BE ANY FASTER THAN ORDER $(n\log_2 n)$. THUS THE SOPHISTICATED ALGORITHMS ARE AS GOOD AS POSSIBLE.

ON THE OTHER HAND, THE SIMPLE ALGORITHMS MAY BE MUCH EASIER TO WRITE QUICKLY AND CORRECTLY. FOR SMALL n, THE PERFORMANCE DIFFERENCE MAY NOT BE SIGNIFICANT. THE ALGORITHM THAT IS SLOWER IN GENERAL (FOR LARGE n) MAY EVEN BE FASTER FOR SMALL n. FOR EXAMPLE, AN ALGORITHM REQUIRING $10n + 25$ OPERATIONS IS ACTUALLY SLOWER THAN AN ALGORITHM REQUIRING $n^2/2$ OPERATIONS FOR $n<23$. (AS n GROWS HOWEVER, THE $10n + 25$ ALGORITHM BECOMES FAR PREFERABLE.)

THE NEXT TWO SLIDES PROVIDE GRAPHS WHICH SHOULD IMPROVE STUDENTS' INTUITIVE GRASP OF THESE IDEAS.

PERFORMANCE OF COMMON SEARCHING AND SORTING ALGORITHMS

	SEARCHING	SORTING
SIMPLE ALGORITHMS	ORDER (n)	ORDER (n^2)
SOPHISTICATED ALGORITHMS	ORDER $(\log_2 n)$	ORDER $(n \log_2 n)$

INSTRUCTOR NOTES

THIS EXAMPLE SHOWS THE DIFFERENCE BETWEEN ORDER (n) AND ORDER ($\log_2 n$) ALGORITHMS. IF $n = 4096$, ($\log_2 n$) IS ONLY 12. THUS WE SEE THAT ORDER (n) IS HUGE WITH RESPECT TO ORDER ($\log_2 n$).

THIS GRAPH ALSO SHOWS WHY ORDER CAN ONLY PLACE ALGORITHMS INTO CLASSES. COMPARED TO $\log_2 n$ AND $n\log_2 n$, WE SEE THAT n AND n/10 ARE QUITE CLOSE. FOR $n = 4096$, WE HAVE

$$n\log_2 n = 49{,}152$$

AND

$$\log_2 n = 12$$

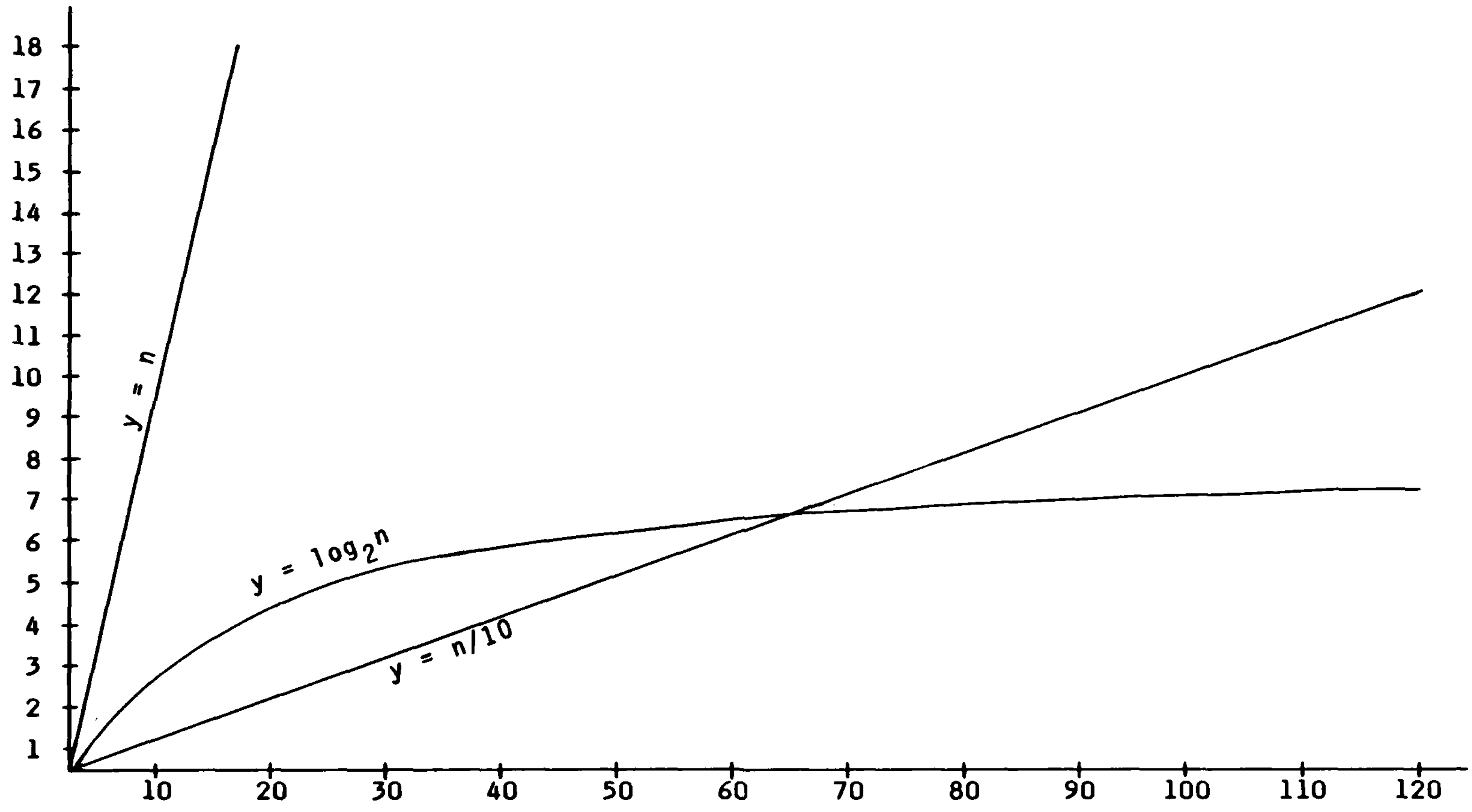
WHILE WE HAVE

$$n = 4096$$

$$n/10 = 410$$

FOR ANY CONSTANTS K_1 AND K_2, $K_1 n$ WILL EVENTUALLY OVERTAKE $K_2 \log_2 n$ FOR LARGE n. DIFFERENCES BETWEEN A 4.0n ALGORITHM AND A 5.0n ALGORITHM CAN BE OVERCOME BY FASTER HARDWARE, BUT AN ORDER ($\log_2 n$) ALGORITHM IS ALWAYS FASTER THAN AN ORDER (n) ALGORITHM FOR LARGE ENOUGH n.

GRAPH OF $y = n, \log_2 n, n/10$



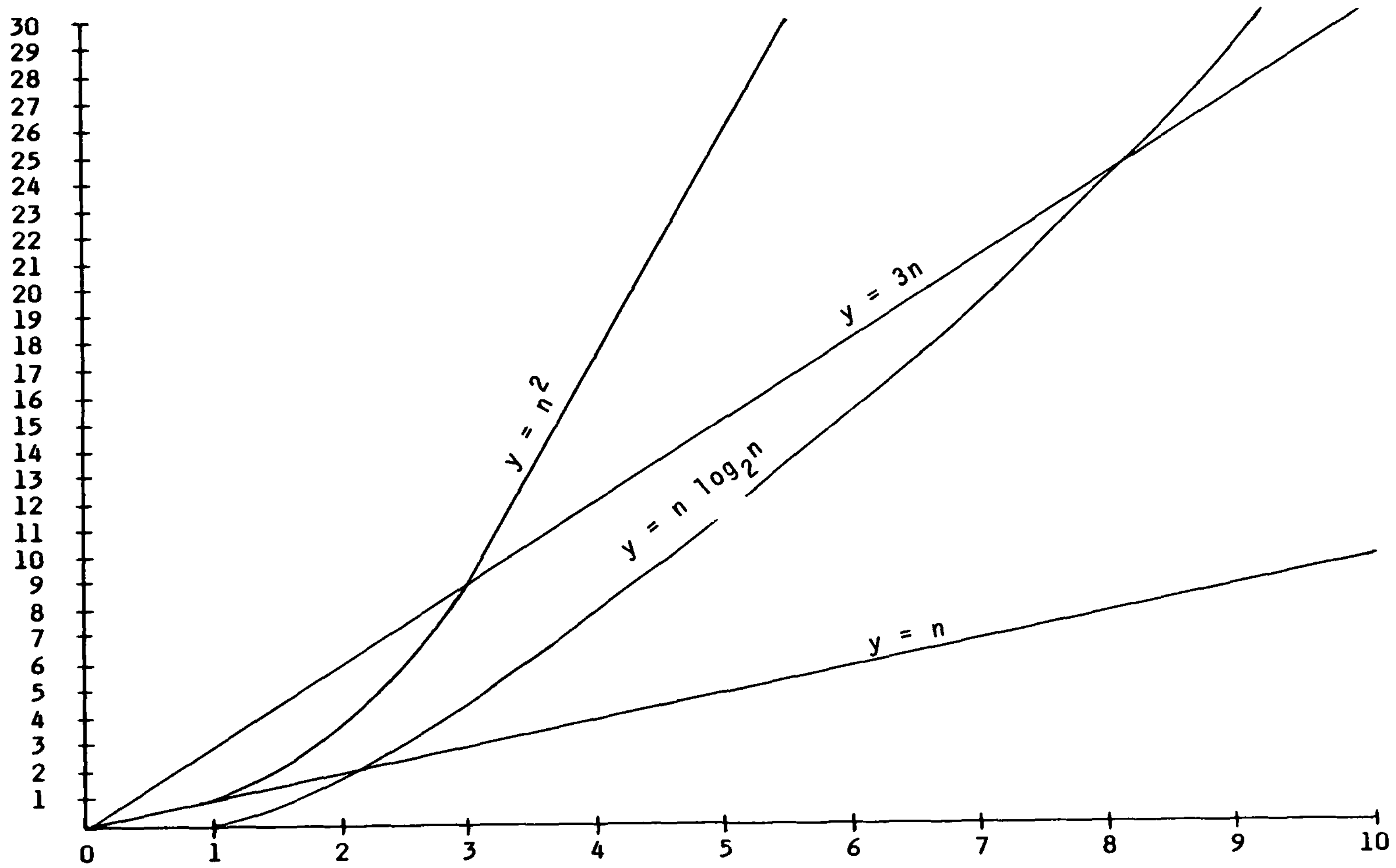
INSTRUCTOR NOTES

THESE GRAPHS SHOW THE DIFFERENCE BETWEEN ORDER (n), ORDER ($n\log_2 n$) AND ORDER (n^2) ALGORITHMS. CLEARLY, ORDER (n^2) IS MUCH WORSE THAN THE OTHERS.

FOR COMPARISON, OF $n = 4096$, THEN $\log_2 n = 12$, $n\log_2 n = 49{,}152$, AND $n^2 = 16{,}777{,}216$.

WHICH ALGORITHM TO CHOOSE DEPENDS ON MANY FACTORS, SUCH AS HOW MANY ELEMENTS ARE TO BE PROCESSED, HOW OFTEN THE PROCESSING IS TO BE PERFORMED, AND HOW MUCH MONEY IS AVAILABLE.

GRAPH OF $y = n^2$, $n\log_2 n$, $3n$, n



INSTRUCTOR NOTES

ANSWERS: AVERAGE NUMBER OF COMPARISONS FOR 13 VALUES: 7

FOR n VALUES: (n + 1)/2

WORST CASE: n

LINEAR SEARCH IS ORDER (n)

AN INEFFICIENT ALGORITHM CAN BE USED IF THE PENALTY FOR USING IT IS NEGLIGIBLE. WE DON'T ALWAYS HAVE TO USE THE BEST ALGORITHM.

SOMETIMES A SIMPLE, QUICKLY WRITTEN, AND EASILY MAINTAINED PROGRAM IS MORE IMPORTANT.

LINEAR SEARCH

1	2	3	4	5	6	7	8	9	10	11	12	13
40	10	59	23	70	34	5	47	42	50	17	30	77

- SCANS LEFT TO RIGHT BY COMPARING FOR EQUALITY
- 2 COMPARES NEEDED TO FIND 10
- 13 COMPARES NEEDED TO FIND 77

- ASSUMING EACH OF THE 13 VALUES IS EQUALLY LIKELY TO OCCUR
  - -- AVERAGE NUMBER OF COMPARES = □

- IN GENERAL, FOR AN n-ELEMENT ARRAY
  - -- AVERAGE NUMBER OF COMPARES = □
  - -- WORST CASE = □

- LINEAR SEARCH IS ORDER (□)

- ACCEPTABLE IF
  - -- NUMBER OF ENTRIES IS SMALL
  - -- SEARCH NOT PERFORMED OFTEN

- NOT ACCEPTABLE IF
  - -- NUMBER OF ENTRIES IS LARGE, OR
  - -- SEARCH PERFORMED OFTEN

INSTRUCTOR NOTES

THIS IS ESSENTIALLY THE PACKAGE PRESENTED EARLIER AS AN EXAMPLE OF DEFAULTS FOR GENERIC FORMAL SUBPROGRAMS.

THIS PACKAGE ALLOWS THE USER TO SPECIFY A Key_Type TO BE SEARCHED FOR AND A Data_Type ASSOCIATED WITH THE Key_Type. A FIXED NUMBER OF ENTRIES IS USED TO HOLD THESE VALUES, WITH A SUITABLE DEFAULT SIZE USED IF THE USER DOES NOT SPECIFY ONE.

TWO PROCEDURES ARE PROVIDED -- ONE TO ASSOCIATE A KEY WITH A DATA VALUE, AND ONE THAT YIELDS THE DATA VALUE ASSOCIATED WITH A KEY. IF A DATA VALUE ALREADY EXISTS FOR A KEY, THEN A REQUEST TO ASSOCIATE A NEW DATA VALUE WITH THE KEY JUST OVERWRITES THE PREVIOUS VALUE. A REQUEST FOR THE DATA ASSOCIATED WITH A NON-EXISTENT KEY YIELDS THE Null_Data VALUE SPECIFIED BY THE INSTANTIATION.

AN EXCEPTION IS PROVIDED IN CASE MORE KEYS ARE SPECIFIED THAN PROVIDED FOR.

THE PACKAGE ALSO ALLOWS THE USER TO SPECIFY A FUNCTION FOR DETERMINING WHETHER TWO KEYS ARE THE SAME. THIS PROVIDES THE USER WITH A WAY OF CONTROLLING WHAT EQUIVALENT KEYS ARE. FOR EXAMPLE, IF Key_Type IS A STRING TYPE, THEN Matching_Keys MIGHT ALLOW TWO STRINGS THAT ONLY DIFFER IN CASE TO BE CONSIDERED THE SAME -- 'BEGIN' AND 'begin' ARE GOOD EXAMPLES.

IF THE USER DOES NOT SPECIFY A VALUE FOR Matching_Keys, THEN THE DEFAULT EQUALITY OPERATOR FOR Key_Type IS USED.

LINEAR SEARCH PACKAGE SPECIFICATION

```
generic

   type Key_Type is private;
   type Data_Type is private;

   Null_Data  : in Data_Type;
   Table_Size : in Integer := 100;

   with function Matching_Keys (Key_1, Key_2 : Key_Type) return Boolean is "=";

package Lookup_Table_Package is

   procedure Update_Data (Key : in Key_Type; Data : in Data_Type);
   procedure Look_Up_Data (Key : in Key_Type; Data : out Data_Type);

   Table_Full_Error : exception;

end Lookup_Table_Package;
```

INSTRUCTOR NOTES

IF THE STUDENTS FIRMLY UNDERSTAND THIS STRATEGY, THE PACKAGE BODY ON THE NEXT THREE SLIDES WILL BE MUCH EASIER TO EXPLAIN.

IN THE EXAMPLE, THERE ARE 13 "REAL" ARRAY ELEMENTS. SINCE THE SCAN PROCEEDS LEFT-TO-RIGHT, IF AN ELEMENT IS "REALLY" IN THE ARRAY IT WILL BE FOUND FIRST AMONG THE FIRST 13 COMPONENTS. THUS 34 IS FOUND AT POSITION 6. IF AN ELEMENT IS NOT "REALLY" IN THE ARRAY, IT WILL BE FOUND FOR THE FIRST TIME IN THE EXTRA SLOT (POSITION 14 IN THIS CASE). THUS 35 IS FOUND AT POSITION 14.

A VALUE IS "REALLY" IN THE ARRAY IF AND ONLY IF IT IS FOUND AT A POSITION EARLIER THAN THE EXTRA SLOT.

THE VALUE IN THE EXTRA SLOT IS SOMETIMES CALLED A SENTINEL.

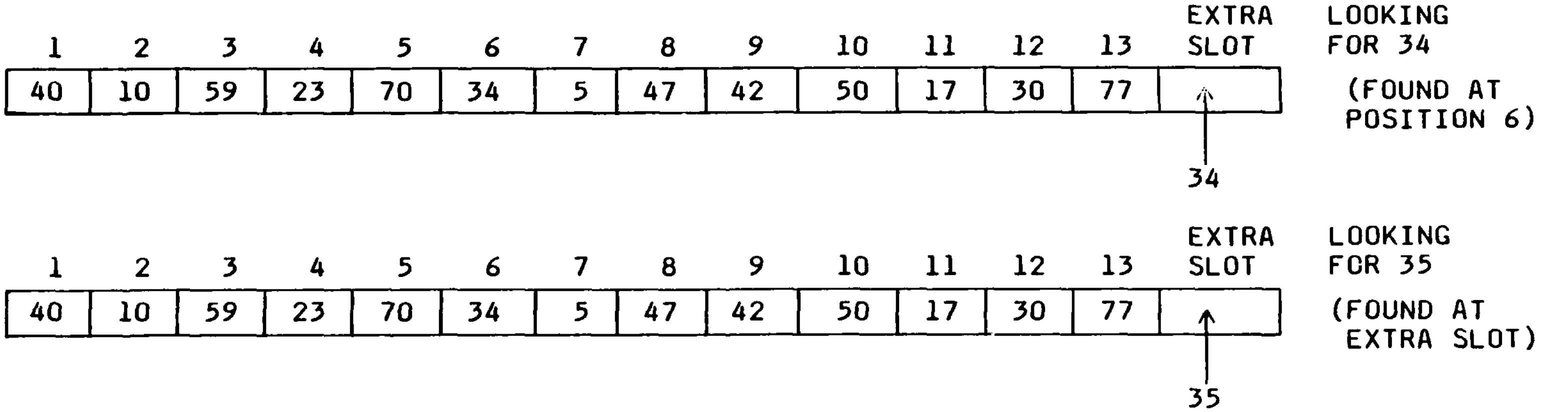
STRATEGY

- APPARENTLY, WE MUST MAKE TWO CHECKS EACH TIME THROUGH THE LOOP THAT SCANS THE ARRAY:

  1. HAVE WE REACHED THE END OF THE ARRAY? (IF SO, NOT FOUND)
  2. IF WE HAVEN'T REACHED THE END, IS THE CURRENT VALUE THE ONE WE ARE LOOKING FOR? (IF SO, FOUND)

- THIS CAN BE AVOIDED BY KEEPING AN EXTRA SLOT AT THE END OF THE ARRAY.

  - BEFORE THE SEARCH, THE VALUE TO BE SEARCHED FOR IS PLACED IN THE EXTRA SLOT.
  - THE VALUE IS THEREFORE GUARANTEED TO BE FOUND SOMEWHERE IN THE ARRAY.
  - AFTER THE LOOP, A SPECIAL CHECK MUST BE PERFORMED TO SEE WHETHER THE VALUE WAS FOUND IN THE EXTRA SLOT.



1	2	3	4	5	6	7	8	9	10	11	12	13	EXTRA SLOT	LOOKING FOR 34
40	10	59	23	70	34	5	47	42	50	17	30	77	↑ 34	(FOUND AT POSITION 6)

1	2	3	4	5	6	7	8	9	10	11	12	13	EXTRA SLOT	LOOKING FOR 35
40	10	59	23	70	34	5	47	42	50	17	30	77	↑ 35	(FOUND AT EXTRA SLOT)

- ADVANTAGES:

  - LOOP IS EASIER TO WRITE CORRECTLY, BECAUSE IT MAY BE ASSUMED THE VALUE WILL BE FOUND
  - LOOP IS FASTER, BECAUSE WE NEED NOT CHECK FOR THE END OF THE ARRAY EACH TIME THROUGH THE LOOP.

INSTRUCTOR NOTES

DO NOT GO INTO MUCH DETAIL ABOUT THE IMPLEMENTATION. EXPLAIN THE DATA STRUCTURES AND GO ON TO Look_Up_Data TO EXPLAIN THE USE OF A SENTINEL (IT'S THE EASIER ONE).

WITHIN THE PACKAGE BODY, WE STORE THE Key AND Data VALUES TOGETHER AS PART OF A SINGLE RECORD, AND MAINTAIN THE VALUES IN A TABLE OF THESE RECORDS.

THE INDEX RANGE OF THE TABLE IS ONE GREATER THAN ACTUALLY NEEDED. THIS ALLOWS US TO STORE THE KEY VALUE AFTER THE LAST VALID TABLE ENTRY, AT Next_Open_Position. THIS ALLOWS US TO USE THE KEY AS A SENTINEL IN THE SEARCH, I.E., A MARKER THAT GUARANTEES WE WILL ALWAYS FIND A MATCH. BECAUSE THE SENTINEL IS PRESENT, WE NEED NOT WORRY ABOUT FALLING OFF THE END OF THE TABLE BEFORE A MATCH IS FOUND.

WE ALWAYS START THE SEARCH AT THE FIRST ENTRY OF THE TABLE AND, BECAUSE OF THE SENTINEL, CONTINUE UNTIL A MATCH IS FOUND. IF THE MATCH OCCURS AT Next_Open_Position, WE KNOW THAT THIS IS A NEW KEY VALUE; OTHERWISE THE KEY ALREADY EXISTS. IN EITHER CASE, WE SIMPLY STORE THE NEW VALUE.

NOTE, HOWEVER, THAT BEFORE WE STORE THE VALUE WE CHECK TO SEE WHETHER THE TABLE IS FULL.

LINEAR SEARCH PACKAGE BODY

```
package body Lookup_Table_Package is

   type Table_Entry_Type is
      record
         Key_Part : Key_Type;
         Data_Part : Data_Type;
      end record;

   Current_Entry_Count : Integer range 0 .. Table_Size := 0;
   Table               : array (1 .. Table_Size + 1) of Table_Entry_Type;

-- CONTINUED ON NEXT SLIDE
```

INSTRUCTOR NOTES

WE AGAIN USE A SENTINEL TO TERMINATE THE SEARCH, BUT THIS TIME WE ALSO STORE THE Null_Data VALUE. THE SEARCHING IS SIMILAR TO WHAT WAS DESCRIBED ON THE PREVIOUS SLIDE, EXCEPT THAT NO DATA IS STORED. NOTE ALSO THAT SINCE THE Null_Data VALUE WAS STORED AS PART OF THE SENTINEL, WE DO NOT NEED TO CONSIDER A SPECIAL CASE.

SUGGEST TO STUDENTS THAT HAVE NEVER SEEN A SENTINEL BEFORE TO TRY REWRITING THIS WITHOUT A SENTINEL. OF COURSE THEY MUST DO IT *OUTSIDE OF CLASS*.

LINEAR SEARCH PACKAGE BODY (Continued)

```
procedure Update_Data (Key : in Key_Type; Data : in Data_Type) is

   Next_Open_Position : constant Integer := Current_Entry_Count + 1;
   Search_Position    : Integer range 1 .. Next_Open_Position := 1;

begin

   -- ADD THE KEY TO THE END OF THE TABLE SO THAT THE SEARCH LOOP MAY ASSUME
   -- IT WILL FIND A MATCHING KEY SOMEWHERE IN THE TABLE:

   Table (Next_Open_Position).Key_Part := Key;

   -- SEARCH FOR THE FIRST MATCHING KEY:

   while not Matching_Keys (Table (Search_Position).Key_Part, Key) loop
      Search_Position := Search_Position + 1:
   end loop;

   -- EXTEND THE TABLE IF THE KEY ADDED ABOVE WAS THE FIRST ONE FOUND, PROVIDED
   -- THERE IS ENOUGH ROOM:

   if Search_Position = Next_Open_Position then
      if Next_Open_Position > Table_Size then
         raise Table_Full_Error;
      else
         Current_Entry_Count := Next_Open_Position;
      end if;
   end if;

   -- UPDATE THE DATA ASSOCIATED WITH THE KEY:

   Table (Search_Position).Data_Part := Data;

end Update_Data;
```

INSTRUCTOR NOTES

LINEAR SEARCH PACKAGE BODY (Continued)

```
procedure Look_Up_Data (Key : in Key_Type; Data : out Data_Type) is

   Search_Position : Integer range 1 .. Current_Entry_Count + 1;

begin

   -- ADD A DUMMY ENTRY WITH NULL DATA TO THE END OF THE TABLE SO THAT THE
   --    SEARCH LOOP MAY ASSUME IT WILL FIND AN ENTRY WITH A MATCHING KEY AND
   --    APPROPRIATE DATA SOMEWHERE IN THE TABLE:

   Table (Current_Entry_Count + 1) := (Key, Null_Data);

   -- SEARCH FOR A MATCHING KEY:

   while not Matching_Keys (Table (Search_Position).Key_Part, Key) loop
      Search_Position := Search_Position + 1;
   end loop;

   -- RETRIEVE THE DATA ASSOCIATED WITH THE FIRST MATCHING KEY:

   Data := Table (Search_Position).Data_Part;

end Look_Up_Data;

end Lookup_Table_Package;
```

INSTRUCTOR NOTES

AN ORDERING FUNCTION "< " WILL BE PROVIDED AS A GENERIC PARAMETER SO INCREASING MEANS INCREASING WITH RESPECT TO THIS GENERIC FORMAL FUNCTION.

ONLY THE EXAMPLES AND PROCEDURE SPECIFICATION WILL BE PRESENTED, SO THE ACTUAL IMPLEMENTATION WILL NOT BE SHOWN IN CLASS.

BINARY SEARCH

- USES A TABLE OF ELEMENTS IN "INCREASING" ORDER

- CONTINUALLY DIVIDES THE TABLE IN HALF, DETERMINING WHICH HALF THE ELEMENT SHOULD BE IN. IF THE CHOSEN HALF IS EMPTY THEN THE ELEMENT DOES NOT EXIST; OTHERWISE IT WILL BE FOUND.

- PERFORMANCE
  - -- AVERAGE PERFORMANCE IS ORDER ($\log_2 n$)
  - -- WORST CASE PERFORMANCE IS ORDER ($\log_2 n$)

- GOOD WHEN
  - -- SET OF ENTRIES IS FIXED, AND
  - -- FREQUENT SEARCHING

- REQUIRES THE ABILITY TO PLACE ELEMENTS "IN ORDER."

INSTRUCTOR NOTES

THE FIRST PAIR OF FIGURES:

- THE FIRST FIGURE SHOWS THE INITIAL STATE OF THE SEARCH. SINCE WE SUBDIVIDE THE INDEX RANGE, WE NEED TO KEEP TRACK OF THE ENDPOINTS (1 AND 13). WE FIND THE MIDPOINT

  $$7 = (1 + 13) / 2$$

  THE MIDPOINT ELEMENT, 40, IS USED TO COMPARE WITH THE ITEM BEING SEARCHED FOR, 30.

- IN THE SECOND FIGURE WE HAVE ADJUSTED THE RIGHT ENDPOINT FROM 13 TO 6, I.E., JUST BEFORE THE MIDPOINT ELEMENT. THE MIDPOINT DIVIDES THE INDEX RANGE IN HALF, SUCH THAT NO ELEMENT IN THE LEFT HALF IS GREATER THAN THE MIDPOINT ELEMENT AND NO ELEMENT IN THE RIGHT HALF IS LESS THAN THE MIDPOINT ELEMENT. SINCE 30   40, THIS MEANS THAT IF IT EXISTS, IT MUST BE IN THE LEFT HALF. THIS IS WHY WE MOVED THE RIGHT ENDPOINT.

THE SECOND PAIR OF FIGURES:

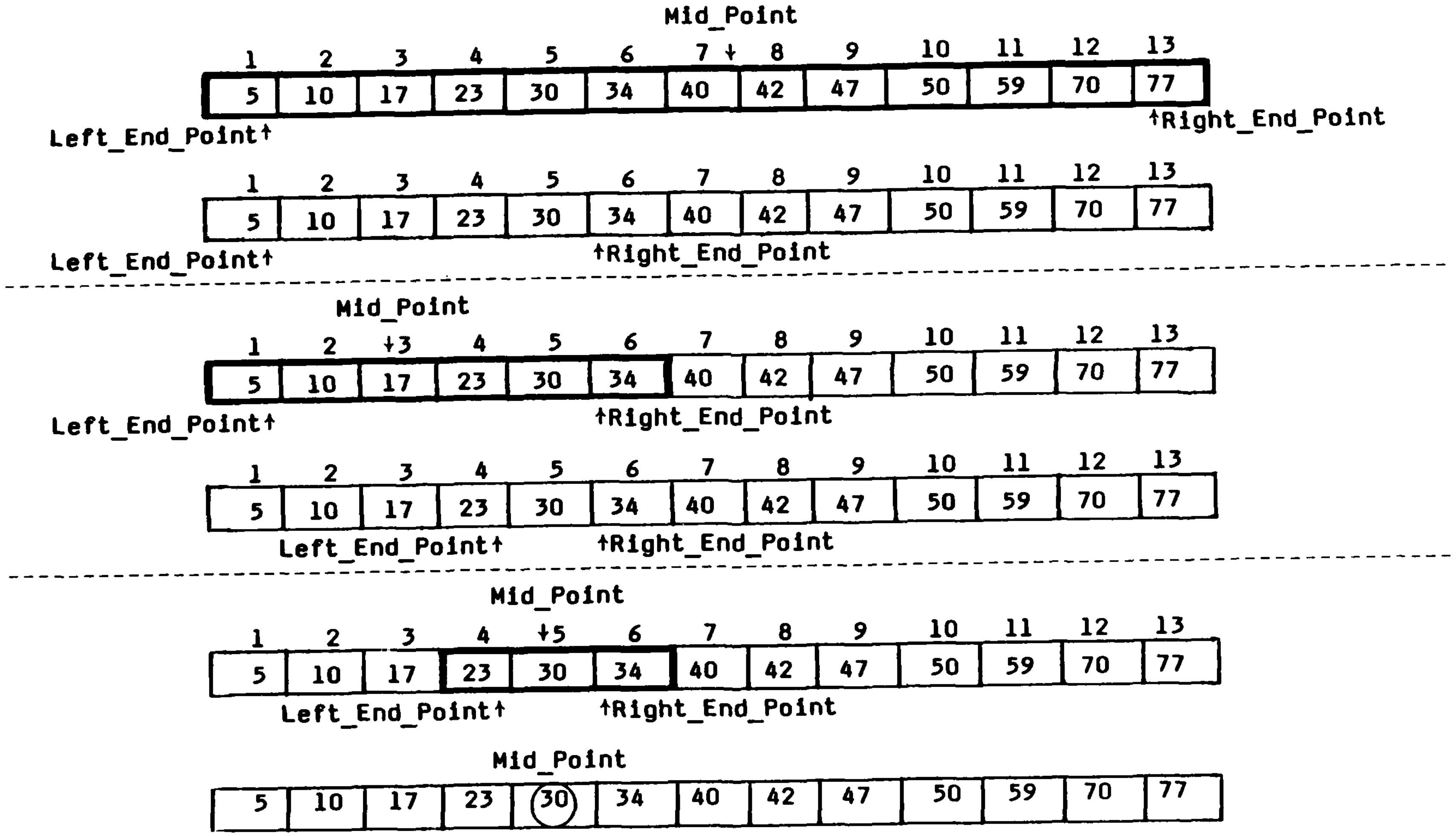
- WE CALCULATE A NEW MIDPOINT, 3, AND COMPARE THE MIDPOINT ELEMENT 17 WITH 30. SINCE 17 < 3C, WE MOVE THE LEFT ENDPOINT PAST THE MIDPOINT.

THE THIRD PAIR OF FIGURES:

- WE CALCULATE A NEW MIDPOINT, 5, AND COMPARE THE MIDPOINT ELEMENT 30 WITH THE SEARCH FOR ELEMENT. SINCE THAT MATCHES, WE ARE HERE. THE SEARCH IS SUCCESSFUL.

BINARY SEARCH EXAMPLE -- A SUCCESSFUL SEARCH

SEARCH FOR 30



```
                                   Mid_Point
           1    2    3    4    5    6    7 ↓  8    9   10   11   12   13
         | 5 | 10 | 17 | 23 | 30 | 34 | 40 | 42 | 47 | 50 | 59 | 70 | 77 |
Left_End_Point↑                                                     ↑Right_End_Point

           1    2    3    4    5    6    7    8    9   10   11   12   13
         | 5 | 10 | 17 | 23 | 30 | 34 | 40 | 42 | 47 | 50 | 59 | 70 | 77 |
Left_End_Point↑                      ↑Right_End_Point
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                 Mid_Point
           1    2   ↓3    4    5    6    7    8    9   10   11   12   13
         | 5 | 10 | 17 | 23 | 30 | 34 | 40 | 42 | 47 | 50 | 59 | 70 | 77 |
Left_End_Point↑                      ↑Right_End_Point

           1    2    3    4    5    6    7    8    9   10   11   12   13
         | 5 | 10 | 17 | 23 | 30 | 34 | 40 | 42 | 47 | 50 | 59 | 70 | 77 |
                Left_End_Point↑      ↑Right_End_Point
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                        Mid_Point
           1    2    3    4   ↓5    6    7    8    9   10   11   12   13
         | 5 | 10 | 17 | 23 | 30 | 34 | 40 | 42 | 47 | 50 | 59 | 70 | 77 |
                Left_End_Point↑      ↑Right_End_Point

                        Mid_Point
         | 5 | 10 | 17 | 23 | 30 | 34 | 40 | 42 | 47 | 50 | 59 | 70 | 77 |
```

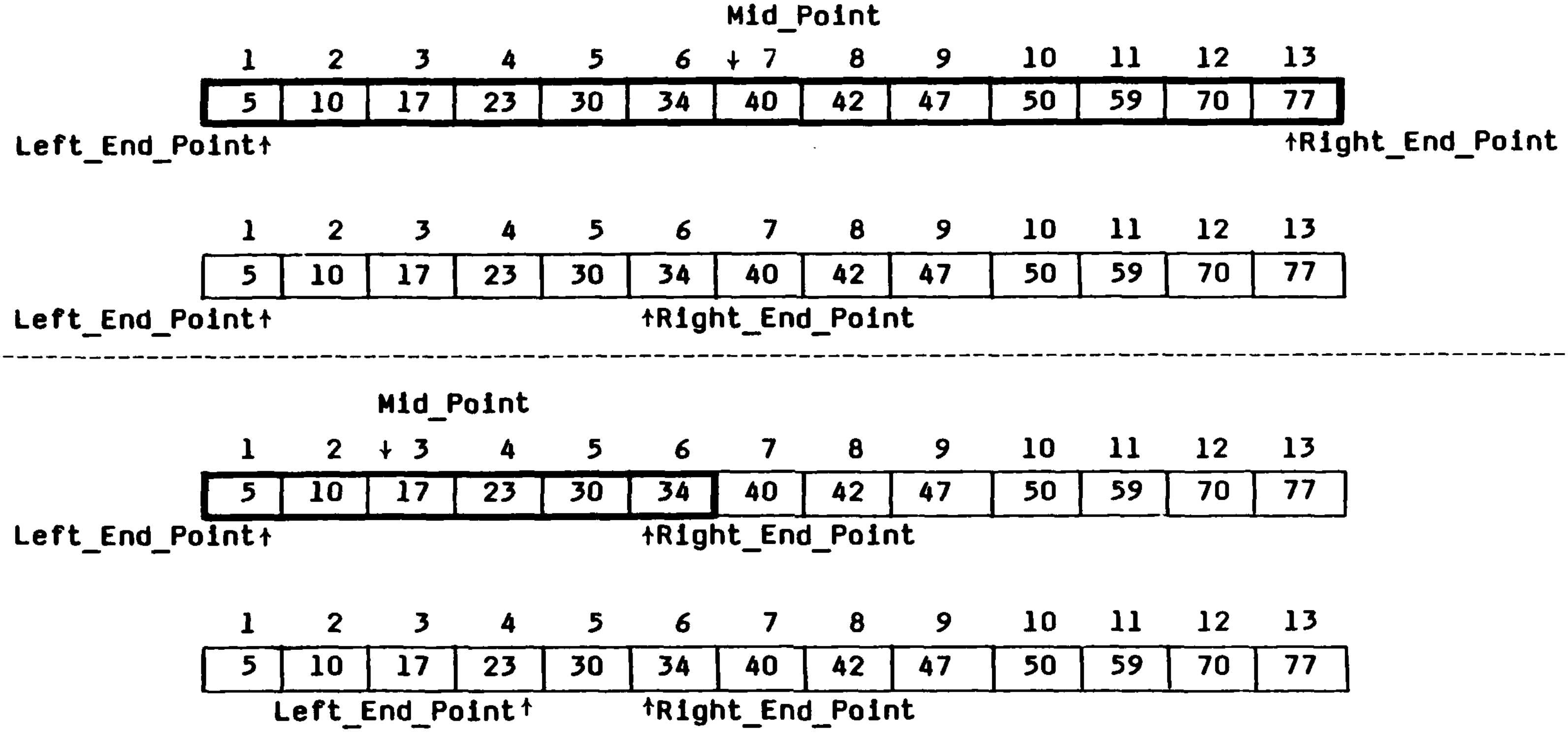
VG 679.2

INSTRUCTOR NOTES

THE FIRST TWO PAIRS OF FIGURES ARE REPEATS OF THOSE ON THE PREVIOUS SLIDE EXCEPT THAT WE ARE SEARCHING FOR 31 INSTEAD OF 30.

BINARY SEARCH EXAMPLE -- AN UNSUCCESSFUL SEARCH

SEARCH FOR 31



```
                                    Mid_Point
           1   2   3   4   5   6  ↓ 7   8   9  10  11  12  13
         | 5 |10 |17 |23 |30 |34 |40 |42 |47 |50 |59 |70 |77 |
Left_End_Point↑                                             ↑Right_End_Point

           1   2   3   4   5   6   7   8   9  10  11  12  13
         | 5 |10 |17 |23 |30 |34 |40 |42 |47 |50 |59 |70 |77 |
Left_End_Point↑                 ↑Right_End_Point
--------------------------------------------------------------------------
                 Mid_Point
           1   2 ↓ 3   4   5   6   7   8   9  10  11  12  13
         | 5 |10 |17 |23 |30 |34 |40 |42 |47 |50 |59 |70 |77 |
Left_End_Point↑                 ↑Right_End_Point

           1   2   3   4   5   6   7   8   9  10  11  12  13
         | 5 |10 |17 |23 |30 |34 |40 |42 |47 |50 |59 |70 |77 |
              Left_End_Point↑   ↑Right_End_Point
```

INSTRUCTOR NOTES

IN THE FIRST PAIR OF FIGURES, WE CALCULATE THE MIDPOINT, 5, AND COMPARE THE MIDPOINT ELEMENT, 30, WITH 31. SINCE 30 < 31, WE ADJUST THE LEFT ENDPOINT PAST THE MIDPOINT.

IN THE SECOND FIGURE, WE COMPUTE THE MIDPOINT, 6, AND COMPARE THE MIDPOINT ELEMENT, 34, WITH 31.

IN THE THIRD FIGURE WE MOVE THE RIGHT END PAST THE MIDPOINT SINCE 34 > 31.

THIS TIME, THE LEFT AND RIGHT ENDPOINTS HAVE CROSSED, SO THE SELECTED HALF INTERVAL IS EMPTY. THE SEARCH IS NOT SUCCESSFUL.

BINARY SEARCH EXAMPLE -- AN UNSUCCESSFUL SEARCH

SEARCH FOR 31 - CONTINUED

Mid_Point

1	2	3	4	↓5	6	7	8	9	10	11	12	13
5	10	17	23	30	34	40	42	47	50	59	70	77

Left_End_Point↑ ↑Right_End_Point

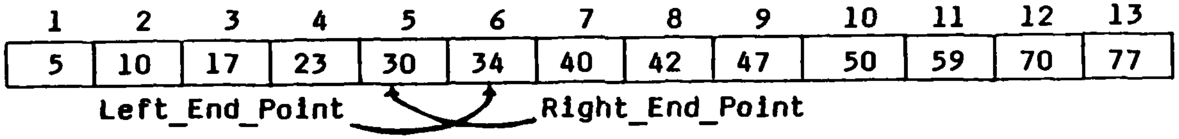
1	2	3	4	5	6	7	8	9	10	11	12	13
5	10	17	23	30	34	40	42	47	50	59	70	77

Left_End_Point↑ ↑Right_End_Point

Mid_Point

1	2	3	4	5	↓6	7	8	9	10	11	12	13
5	10	17	23	30	34	40	42	47	50	59	70	77

Left_End_Point↑ ↑Right_End_Point



1	2	3	4	5	6	7	8	9	10	11	12	13
5	10	17	23	30	34	40	42	47	50	59	70	77

Left_End_Point Right_End_Point

INSTRUCTOR NOTES

THE THIRD GENERIC PARAMETER SPECIFIES THE CRITERIA BY WHICH TABLE ENTRIES ARE ORDERED. FOR ANY TWO VALUES I AND J IN POSITIVE RANGE Table'Range, IT MUST BE THE CASE THAT I < J (ACCORDING TO PREDEFINED " < " FOR TYPE Integer), IF AND ONLY IF Table (I) < Table (J) (ACCORDING TO THE VERSION OF " > " FOR Data_Type SPECIFIED IN THE INSTANTIATION).

THE NEXT EXAMPLE SHOWS THE USE OF Table_Index SO MENTION ITS EXISTENCE AND WHAT WE USE IT FOR.

BINARY SEARCH SPECIFICATION

```
generic

   type Data_Type is private;
   type Table_Type is array (Positive range < >) of Data_Type;

   with function " < " (Data_1, Data_2 : Data_Type) return Boolean is " < ";

-- THIS PROCEDURE PERFORMS A BINARY SEARCH ON Table, LOOKING FOR Data.  IF THE
--    SEARCH IS SUCCESSFUL, THEN Found IS SET TO True; OTHERWISE IT IS SET TO
--    False.  IN THE CASE OF A SUCCESSFUL SEARCH, Table_Index IS THE INDEX OF
--    THE ENTRY IN Table THAT EQUALS Data.  IF THE SEARCH IS NOT SUCCESSFUL,
--    THEN Table_Index IS THE INDEX AFTER WHICH Data SHOULD BE STORED.

procedure Binary_Search
   (Table       : in Table_Type;
    Data        : in Data_Type;
    Table_Index : out Positive;
    Found       : out Boolean);
```

INSTRUCTOR NOTES

THE WORST CASE CAN BE PREVENTED BY VARIOUS MODIFICATIONS TO THE BASIC ALGORITHM.

WE WILL GIVE A SIMPLE EXAMPLE AND THEN SHOW THE BASIC ALGORITHM. WE DESCRIBE THE VARIOUS MODIFICATIONS THAT CAN BE MADE, BUT POINT TO THE LITERATURE FOR DETAILS.

SEARCH TREES

- ALLOW DYNAMIC GROWTH OF THE SET OF VALUES TO BE SEARCHED
- EASY TO INSERT NEW VALUES
- PERFORMANCE

  AVERAGE CASE = ORDER ($\log_2 n$)

  WORST CASE = ORDER (n)

- GOOD WHEN:

  -- SET OF ENTRIES NOT FIXED

  -- NUMBER OF ENTRIES LARGE

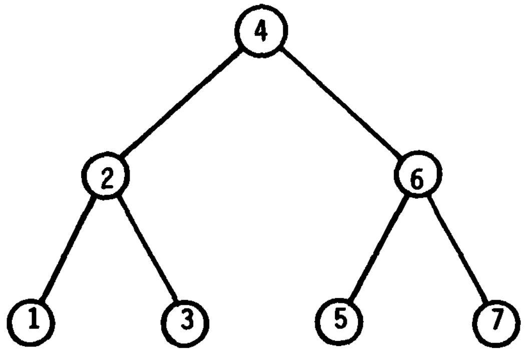
  -- FREQUENT SEARCHING

INSTRUCTOR NOTES

THE EXAMPLE TREE WILL BE CONSTRUCTED IN THE NEXT SLIDE.

SHOW HOW THE TOP 3 NODES OBEY THE DEFINITION OF A SEARCH TREE.

SEARCH TREES



- A SEARCH TREE IS A TREE SUCH THAT FOR ANY NODE X, ALL VALUES IN THE LEFT SUBTREE ARE LESS THAN OR EQUAL THE VALUE AT X AND THE VALUE AT X IS LESS THAN OR EQUAL TO ALL VALUES IN THE RIGHT SUBTREE.

INSTRUCTOR NOTES

IN THE FIRST ROW OF TREES, WE SIMPLY ADD THE VALUE 4 TO A NULL TREE.

IN THE SECOND ROW OF TREES WE WANT TO ADD THE VALUES 2 AND 6 TO THE TREE. SINCE 2 IS LESS THAN 4 WE ADD 2 AS THE LEFT CHILD. SINCE 6 IS GREATER THAN 4 WE ADD 6 AS THE RIGHT CHILD.

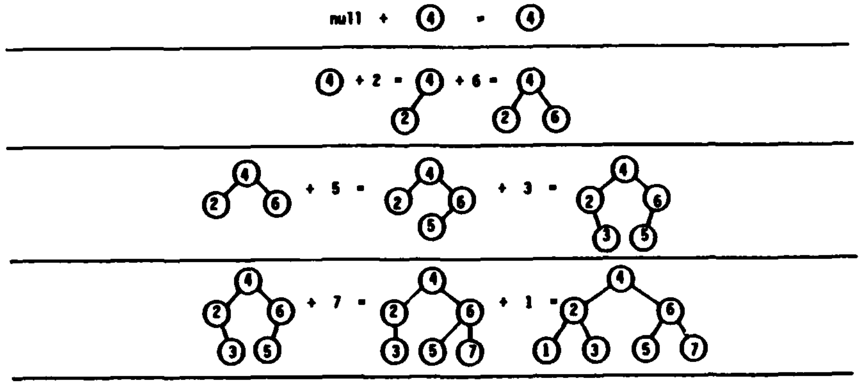
IN THE THIRD ROW OF TREES, WE WANT TO ADD THE VALUES 5 AND 3 TO THE TREE. SINCE 5 IS GREATER THAN THE ROOT VALUE 4, WE WANT TO ADD 5 AS THE RIGHT CHILD, BUT IT IS ALREADY OCCUPIED. THEREFORE, WE CONTINUE WITH THE RIGHT SUBTREE. SINCE 5 IS LESS THAN THE ROOT VALUE 6 OF THE SUBTREE, WE ADD 5 AS THE LEFT CHILD OF THE SUBTREE NODE. SIMILARLY, SINCE 3 IS LESS THAN THE ROOT VALUE 4, WE WANT TO ADD 3 AS THE LEFT CHILD OF THE ROOT. SINCE IT IS OCCUPIED, WE CONTINUE WITH THE LEFT SUBTREE. SINCE 3 IS GREATER THAN THE ROOT VALUE 2 OF THE SUBTREE, WE ADD 3 AS THE RIGHT CHILD OF THE SUBTREE NODE.

IN THE FOURTH ROW OF TREES, WE ARE JUST ADDING TWO MORE VALUES.

THE FINAL FIGURE ILLUSTRATES HOW WE SEARCH A TREE. WE SEARCH THE TREE FOR THE VALUE 3. STARTING WITH THE ROOT, 3 IS LESS THAN THE ROOT VALUE 4, SO WE CONTINUE WITH THE LEFT SUBTREE. SINCE 3 IS GREATER THAN THE ROOT VALUE 2 OF THE SUBTREE, WE CONTINUE WITH THE RIGHT SUBTREE. NOW THE NODE VALUE IS 3, SO THE SEARCH IS SUCCESSFUL.

IN GENERAL, WE START WITH THE ROOT AND PROCEED WITH THE LEFT OR RIGHT SUBTREE, DEPENDING ON WHETHER THE VALUE BEING SEARCHED IS LESS THAN OR GREATER THAN THE ROOT VALUE. SEARCHING TERMINATES WHEN EITHER THE ROOT VALUE OF A SUBTREE MATCHES THE SEARCHED FOR VALUE, IN WHICH CASE THE VALUE IS IN TREE, OR WHEN WE GO TO FOLLOW A SUBTREE, WE FIND IT TO BE NULL, IN WHICH CASE THE SEARCHED FOR VALUE IS NOT IN THE TREE.

WALK THE CLASS THROUGH THE CONSTRUCTION OF THE TREE:



SEARCH TREES -- A DECEPTIVELY NICE EXAMPLE

BUILD A TREE BY ADDING ENTRIES IN THE ORDER ④,②,⑥,⑤,③,⑦,①:

INSTRUCTOR NOTES

WE ALLOW "<" TO BE SPECIFIED. WE NEED THIS TO DETERMINE WHETHER TO SEARCH THE LEFT OR RIGHT SUBTREE.

THE ONLY DIFFERENCE BETWEEN THIS PACKAGE AND THE Linear_Search PACKAGE ARE THAT

--   LINEAR SEARCH REQUIRED A TABLE SIZE

--   LINEAR SEARCH USED Matching_Keys FOR EQUALITY WHILE THIS PACKAGE USES "<".

WE GIVE A RECURSIVE IMPLEMENTATION.

Lookup_Package SPECIFICATION

```
generic

   type Key_Type is private;
   type Data_Type is private;

   Null_Data : in Data_Type;

   with function " < " (Key_1, Key_2 : Key_Type) return Boolean is "<";

package Lookup_Table_Package is

   procedure Update_Data (Key : in Key_Type; Data : in Data_Type);
   procedure Look_Up_Data (Key : in Key_Type; Data : out Data_Type);

end Lookup_Table_Package;
```

INSTRUCTOR NOTES

NOTICE HOW CLOSELY THE NODE RECORD MODELS AN ACTUAL TREE. IT CONTAINS THE NODE DATA (Key_Part AND Data_Part), AND A LEFT AND RIGHT SUBTREE. THE SUBTREES ARE ACCESSED VIA THE ACCESS TYPE TREE.

NOTICE THAT THE ROOT OF THE TREE IS INITIALIZED TO NULL, SINCE THE TREE IS EMPTY. WE ALSO USE NULL TO INDICATE IF A SUBTREE IS EMPTY.

Update_Data MAKES USE OF AN AUXILIARY PROCEDURE, Update_Data_In. REMEMBER THAT THE AUXILIARY PROCEDURE IS VISIBLE WITHIN THE PACKAGE BODY ONLY -- THE PACKAGE USER DOES NOT KNOW IT EXISTS.

Update_Data_In FOLLOWS THE PROCEDURE WE OUTLINED IN THE EXAMPLE. AS WE VISIT EACH NODE (BY CALLING Update_Data_In), WE COMPARE THE KEY VALUE WITH THE Key_Part OF THE NODE. IF THE NODES ARE NOT EQUAL, THEN WE PROCEED WITH EITHER THE LEFT OR RIGHT SUBTREE (BY CALLING Update_Data_In WITH Left_Subtree OR Right_Subtree, RESPECTIVELY). IF WE REACH AN EMPTY SUBTREE, THEN KEY IS NOT IN THE TREE, SO WE ADD IT AND DATA AS A NODE IN THE TREE AT THAT POINT. NOTE THAT THIS IS ACCOMPLISHED IN THE PROCEDURE BY ALLOCATING A NEW INSTANCE OF Node_Type AND ASSIGNING IT TO THE ACCESS TYPE PARAMETER Tree. SINCE Tree HAS in out MODE, THIS LINKS THE NODE INTO THE TREE.

THE COMPONENTS Key_Part AND Data_Part COLLECTIVELY CORRESPOND TO THE COMPONENT NAMED Data_Part IN THE INTRODUCTORY DISCUSSION OF TREES.

THE NON-RECURSIVE VERSION (SEE NEXT INSTRUCTOR NOTE) OF Update_Data IS CONSIDERABLY MORE COMPLICATED THAN THE RECURSIVE VERSION. WHAT IS NOT APPARENT IN THE RECURSIVE VERSION IS THAT EACH PROCEDURE CALL KEPT TRACK OF THE NODE POSITION IN THE TREE, AND WHETHER A LEFT OR A RIGHT SUBTREE WAS BEING PROCESSED. THE Tree PARAMETER BEING PART OF THE PREVIOUS NODE ALLOWS US TO ACCOMPLISH THIS. WHEN WE FIND THAT THE TREE IS NULL, WE JUST ASSIGN A NEW NODE TO IT.

WITHOUT RECURSION WE MUST STILL KEEP TRACK OF THIS. IN THE NON-RECURSIVE VERSION WE DISTRIBUTE THE CHECK FOR A NULL TREE. BEFORE FOLLOWING A SUBTREE WE ALWAYS CHECK IF THE SUBTREE IS EMPTY. IF IT IS, WE ADD A NEW NODE AND EXIT. NOTICE WE DO NOT ADVANCE THE SUBTREE POINTER UNTIL WE KNOW THAT THE SUBTREE IS NOT NULL.

Lookup_Table_Package BODY

```
package body Lookup_Table_Package is

   type Node_Type;
   type Tree_Type is access Node_Type;
   type Node_Type is
      record
         Key_Part                      : Key_Type;
         Data_Part                     : Data_Type;
         Left_Subtree, Right_Subtree : Tree_Type;
      end record;

   Root : Tree_Type := null;

   -- CONTINUED ON NEXT SLIDE
```

INSTRUCTOR NOTES

IN CASE A STUDENT ASKS WHY RECURSIVE AND NOT NON-RECURSIVE, A NON-RECURSIVE VERSION FOLLOWS FOR YOUR READING, ALONG WITH SOME NOTES.

```
procedure Update_Data (Key : in Key_Type; Data : in Data_Type) is
   Tree : Tree_Type := Root;
begin
   if Tree = null then
      Root := new Node_Type'(Key, Data, null, null);
      return;
   end if;
   loop
      if Key < Tree.Key_Part then
         if Tree.Left_Subtree /= null then
            Tree := Tree.Left_Subtree;
         else
            Tree.Left_Subtree := new Node_Type (Key, Data, null, null);
            return
         end if;
      elsif Tree.Key_Part < Key then
         if Tree.Right_Subtree /= null then
            Tree := Tree.Right_Subtree;
         else
            Tree.Right_Subtree := new Node_Type (Key, Data, null, null);
            return;
         end if;
      else
         Tree.Data_Part := Data;
         return;
      end if;
   end loop;
end Update_Data;
```

Lookup_Table_Package BODY (Continued)

```
procedure Update_Data_In
   (Tree : in out Tree_Type;
    Key  : in Key_Type;
    Data : in Data_Type) is
begin -- Update_Data_In
   if Tree = null then
      Tree := new Node_Type (Key, Data, null, null);
   elsif Key < Tree.Key_Part then
      Update_Data_In (Tree.Left_Subtree, Key, Data);
   elsif Tree.Key_Part < Key then
      Update_Data_In (Tree.Right_Subtree, Key, Data);
   else
      Tree.Data_Part := Data;
   end if;
end Update_Data_In;

procedure Update_Data (Key : in Key_Type; Data : in Data_Type) is
begin -- Update_Data
   Update_Data_In (Root, Key, Data);
end Update_Data;

-- CONTINUED ON NEXT SLIDE
```

INSTRUCTOR NOTES

Look_Up_Data_In IS SIMILAR TO Update_Data_In EXCEPT THAT IT RETURNS DATA TO THE USER, RATHER THAN CREATING NEW NODES.

BOTH Look_Up_Data_In AND Update_Data_In HAVE THE SAME TERMINATION CONDITIONS FOR THE RECURSION -- IF THE TREE IS null, OR THE SEARCHED-FOR ENTRY IS FOUND. ALWAYS REMEMBER TO HAVE TERMINATION CONDITIONS IN RECURSIVE SUBPROGRAMS.

Lookup_Table_Package BODY (Continued)

```
   procedure Look_Up_Data_In(Tree: in Tree_Type; Key: in Key_Type; Data: out Data_Type) is
   begin -- Look_Up_Data_In
      if Tree = Null then
         Data := Null_Data;
      elsif Key   Tree.Key_Part then
         Look_Up_Data_In (Tree.Left_Subtree, Key, Data);
      elsif Tree.Key_Part < Key then
         Look_Up_Data_In (Tree.Right_Subtree, Key, Data);
      else
         Data := Tree.Data_Part;
      end if;
   end Look_Up_Data_In;

   procedure Look_Up_Data (Key : in Key_Type; Data : out Data_Type) is
   begin -- Look_Up_Data
      Look_Up_Data_In (Root, Key, Data);
   end Look_Up_Data;

end Lookup_Table_Package;
```

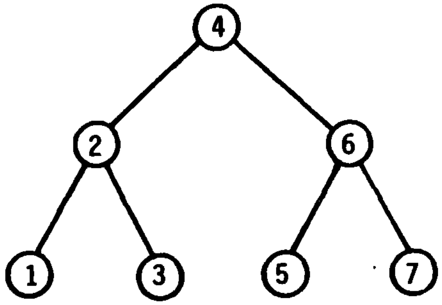
INSTRUCTOR NOTES

NOTE WE BUILT THIS TREE IN THE EXAMPLE.

IF WHENEVER WE BUILD A SEARCH TREE, WE OBTAINED A "NICE" TREE LIKE THIS, WE WOULD KNOW THAT THE PERFORMANCE OF THESE TREES IS ORDER ($\log_2 n$), I.E., ORDER (THE HEIGHT OF THE TREE).

WHILE THE ORDER IN WHICH THESE VALUES WERE INSERTED INTO THE TREE MAY SEEM RANDOM, THE ORDER IS ACTUALLY CONTRIVED TO CREATE A NICE TREE. THE NEXT FIGURE SHOWS THAT CAN HAPPEN IN THE WORST CASE.

A NICE SEARCH TREE

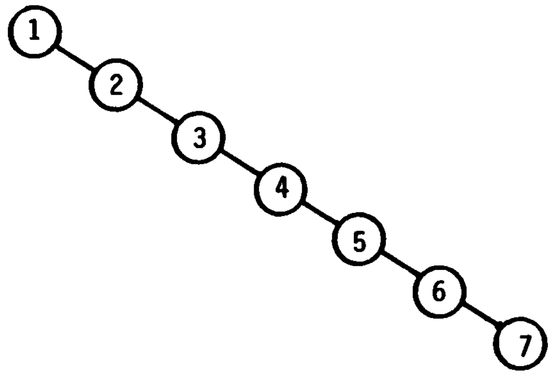


NODES ADDED IN THE ORDER

4, 2, 6, 5, 3, 7, 1

INSTRUCTOR NOTES

RESULT:



SEARCHING THIS TREE YIELDS ORDER (N) PERFORMANCE.

THIS IS ESSENTIALLY A LINEAR SEARCH.

THIS WORST CASE ARISES NOT FROM A RANDOM INSERTION ORDER, BUT FROM INSERTION OF NODES ALREADY IN ASCENDING (OR ALREADY IN DESCENDING) ORDER.

WE WOULD LIKE TO STAY AS CLOSE TO "NICE" TREES AS POSSIBLE.

A BAD SEARCH TREE

ADD ENTRIES IN THE ORDER①,②,③,④,⑤,⑥,⑦:

INSTRUCTOR NOTES

THIS IS NOT A CONTRIVED EXAMPLE. THE WORST CASE ARISES WHEN ENTRIES ARE ADDED IN PERFECT ASCENDING OR DESCENDING ORDER. IT IS EASY TO ENVISION APPLICATIONS IN WHICH THIS IS PRECISELY WHAT WE CAN EXPECT TO OCCUR.

THE NEXT SLIDE PRESENTS A DIFFERENT APPROACH TO MAINTAINING A BALANCED SEARCH TREE.

THE NEED FOR BALANCED TREES

- THE TREE ON THE PREVIOUS SLIDE IS ESSENTIALLY A LINEAR LIST.
  - LENGTH IS EQUAL TO NUMBER OF ENTRIES.
  - THIS IS ESSENTIALLY A LINEAR SEARCH, WITH ORDER (n) PERFORMANCE.

- TO ACHIEVE ORDER ($\log_2 n$) PERFORMANCE, WE MUST KEEP THE TREE "BALANCED".
  - LEFT AND RIGHT SUBTREES OF A NODE SHOULD BE APPROXIMATELY THE SAME SIZE.
  - RESULTING TREE IS SHALLOW AND WIDE, NOT DEEP AND NARROW.
  - SHORTER AVERAGE DISTANCE FROM ROOT MEANS SHORTER SEARCH TIME.

- SOME INSERTION ALGORITHMS RE-SHAPE THE SEARCH TREE TO KEEP IT BALANCED.
  - MAINTAINING PERFECT BALANCE IS TOO EXPENSIVE TO BE WORTHWHILE.
  - ALGORITHMS TO MAINTAIN GOOD, BUT NOT OPTIMAL, BALANCE CAN BE WORTHWHILE.

INSTRUCTOR NOTES

NOTICE THAT BOXES ARE USED FOR LEAVES. THIS SUGGESTS THAT THE LEAVES MIGHT NOT CONTAIN THE SAME INFORMATION AS THE NON-LEAF NODES.

AS AN EXAMPLE, THE TREE SHOWN MIGHT BE USED BY A TEXT EDITOR. THE NON-LEAF NODES COULD CONTAIN LINE NUMBERS AND THE LEAVES CONTAIN POINTERS TO PRIMARY AND SECONDARY MEMORY STORAGE FOR LINES OF TEXT. FOR EXAMPLE, NODE (5, 10) MIGHT MEAN THAT STORAGE INFORMATION FOR LINES 1-5 IS IN ITS LEFTMOST CHILD; STORAGE INFORMATION FOR LINES 6-10 IS IN ITS SECOND CHILD, AND STORAGE INFORMATION FOR LINES 11-20 IS IN ITS RIGHTMOST CHILD.

2-3 TREE

- A KIND OF SEARCH TREE THAT IS EASY TO KEEP BALANCED.

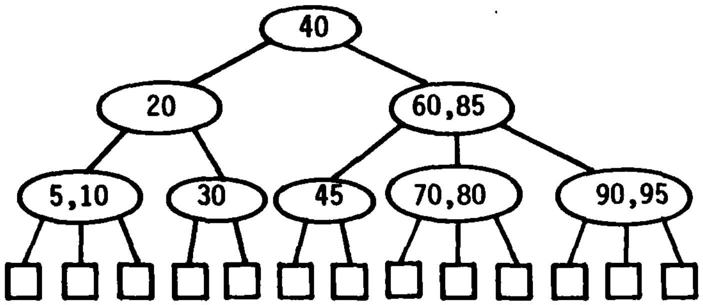
- A TREE IS A 2-3 TREE IF AND ONLY IF

  -- EVERY NON-LEAF NODE EITHER

  i. CONTAINS ONE KEY AND HAS 2 CHILDREN, OR

  ii. CONTAINS TWO KEYS AND HAS 3 CHILDREN, AND

  -- EVERY PATH FROM THE ROOT TO A LEAF HAS THE SAME LENGTH



- ONE OF THE BEST ORDER ($\log_2 n$) SEARCHES AND ONE OF THE MOST SPACE EFFICIENT.
- SEE SECTION 4.9 OF AHO, HOPCROFT, AND ULLMAN FOR DETAILS.

INSTRUCTOR NOTES

THE HASHING FUNCTION IS APPLIED TO A KEY TO YIELD AN INDEX INTO THE TABLE.

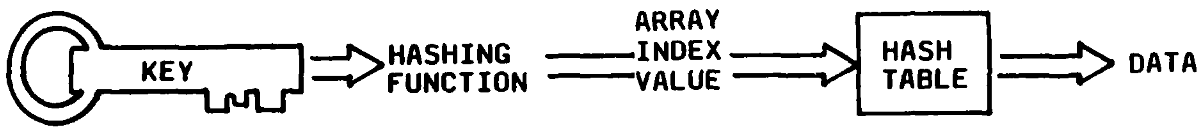
IF WE ARE HASHING NAMES THAT CAN BE UP TO 10 CHARACTERS IN LENGTH INTO A HASH TABLE OF 1000 ENTRIES THEN THE HASH FUNCTION MAPS 26^{10} POSSIBLE VALUES INTO 10^{3} DIFFERENT INDEXES.

HASH FUNCTION MUST PROVIDE GOOD SCATTERING (DISTRIBUTION) OF KEYS ONTO THE RANGE.

WHEN COLLISIONS OCCUR FREQUENTLY, PERFORMANCE DECREASES.

CANNOT COMPLETELY ELIMINATE COLLISIONS SO WE MUST ALSO CONSIDER HOW TO HANDLE THEM.

HASHING



- FASTEST CLASS OF LOOKUP ROUTINES FOR FIXED-SIZE TABLES.

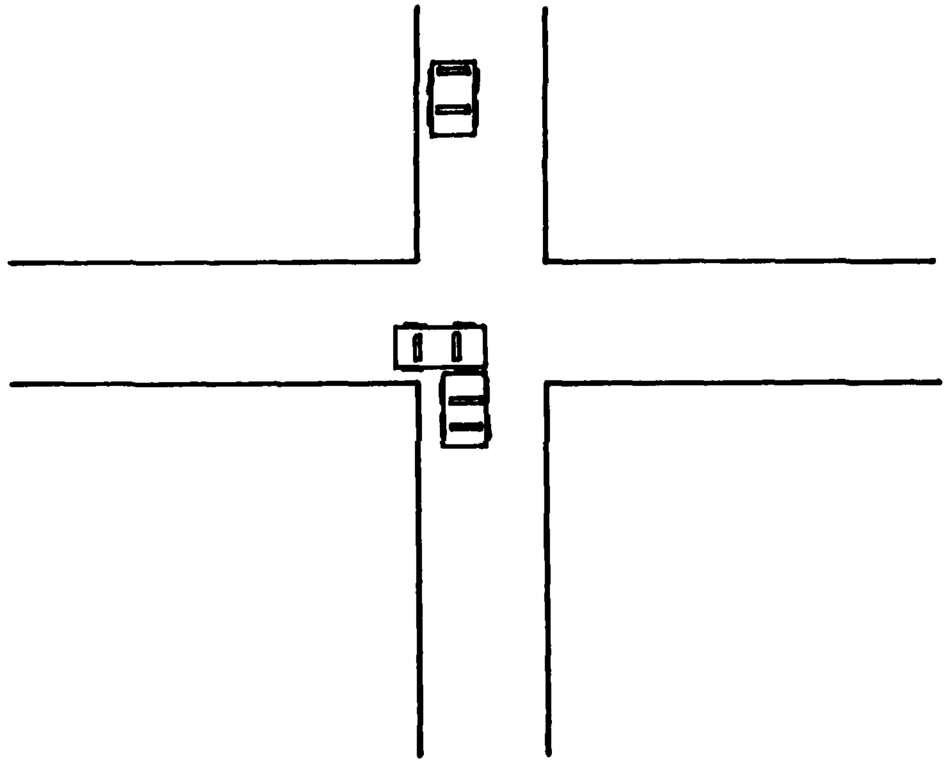
- PROVIDES "ALMOST DIRECT ACCESS."

- A "HASHING FUNCTION" IS AN ARBITRARY FUNCTION THAT TAKES A KEY AS A PARAMETER AND RETURNS SOME INTEGER THAT CAN BE USED AS AN INDEX VALUE FOR AN ARRAY. A HASHING FUNCTION MAPS A LARGE KEY RANGE INTO A SMALL RANGE OF ARRAY INDICES (THE HASHING FUNCTION IS MANY-TO-ONE.)

- PROBLEMS WITH COLLISIONS WHEN SEVERAL KEYS ARE MAPPED TO THE SAME ARRAY INDEX.

INSTRUCTOR NOTES

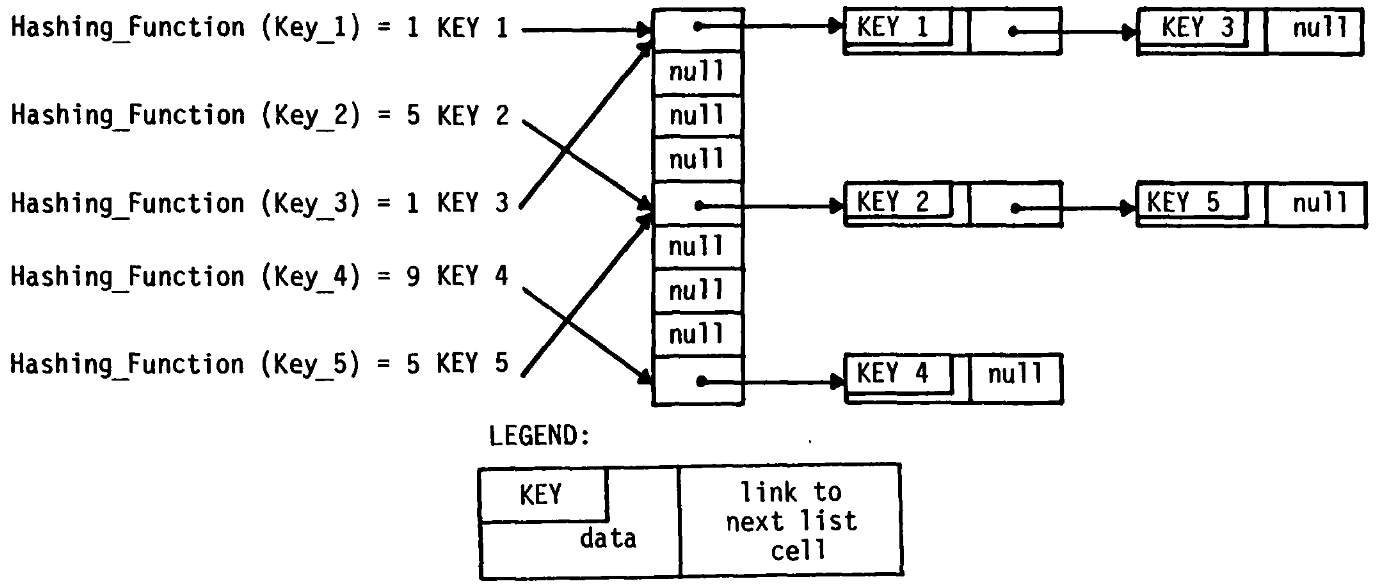
COLLISIONS



INSTRUCTOR NOTES

TWO EXAMPLES FOLLOW THAT ILLUSTRATE WHAT THE DIFFERENCE BETWEEN A GOOD AND BAD HASHING FUNCTION CAN MEAN.

REPRESENTING COLLISIONS IN A HASH TABLE -- CHAINING


Hashing_Function (Key_1) = 1 KEY 1
Hashing_Function (Key_2) = 5 KEY 2
Hashing_Function (Key_3) = 1 KEY 3
Hashing_Function (Key_4) = 9 KEY 4
Hashing_Function (Key_5) = 5 KEY 5
null
null
null
null
null
null
KEY 1
KEY 3
null
KEY 2
KEY 5
null
KEY 4
null
LEGEND:
KEY
data
link to next list cell

- HASH FUNCTIONS ARE MANY-TO-ONE.
- EACH ENTRY IN THE HASH TABLE IS ACTUALLY A LINKED LIST. THE LINKED LIST CONTAINS ALL THE ELEMENTS THAT MAP TO THE SAME VALUE.
- CHECKING A HASH TABLE FOR A GIVEN VALUE REQUIRES SEARCHING A LINKED LIST.
  - -- IF TOO MANY COLLISIONS THEN THE LISTS ARE TOO LONG.
  - -- IF THE LISTS ARE TOO LONG, PERFORMANCE WILL DEGRADE.
- GOOD HASHING FUNCTIONS MINIMIZE COLLISIONS.

INSTRUCTOR NOTES

THIS HASHING FUNCTION SIMPLY SUMS UP THE POSITION NUMBERS OF THE CHARACTERS IN THE STRING, THEN EVALUATES IT mod THE Table_Size AND THEN ADDS 1. THIS KEEPS THE FUNCTION RESULT WITHIN THE Table_Range.

FOR A LARGE STRING, SUMMING UP THE ORDINAL VALUES FIRST, MIGHT EXCEED THE PROCESSOR'S ARITHMETIC CAPABILITY. THEREFORE, WE APPLY THE mod OPERATION AT EACH ITERATION. SINCE

$$(A + B) \bmod C = ((A \bmod C) + B) \bmod C$$

THIS DOES NOT CHANGE THE RESULT.

NOTICE THAT THE COMPUTATION Hash_Value + Char_Value CANNOT BE RESTRICTED TO THE RANGE 1 .. Table_Size. THEREFORE, WE DEFINE A TYPE Hash_Range_Type WHICH PROVIDES A RANGE LARGE ENOUGH TO PERFORM THE COMPUTATION AND DEFINE Table_Range_Type AS A SUBTYPE OF Hash_Range_Type.

WE SEE FOUR EXAMPLES OF THE VALUE RETURNED BY THIS HASH FUNCTION. WE SEE A PROBLEM, HOWEVER, SINCE STRINGS THAT ARE SIMILAR ARE HASHED TO THE SAME VALUE. A HASH FUNCTION SHOULD DISTRIBUTE THE KEYS FAIRLY EVENLY ACROSS THE TABLE RANGE.

A SIMPLE HASHING FUNCTION FOR STRINGS

```
   ...
Table_Size : constant := ...;
   ...
type Hash_Range_Type is 0 .. Table_Size + Character'Pos (Character'Last);
subtype Table_Range_Type is Hash_Range_Type range 1 .. Table_Size;
   ...
function Simple_Hash (Key : in String) return Table_Range_Type is

   Hash_Value : Hash_Range_Type := 0;
   Char_Value : Hash_Range_Type;

begin -- Simple_Hash

   for I in Key'Range loop
      Char_Value := Character'Pos (Key (I));
      Hash_Value := (Hash_Value + Char_Value) mod Table_Size;
   end loop;
   return Hash_Value + 1;

end Simple_Hash;
   ...
```

```
Simple_Hash ("A2") = ((65 + 50) mod 97) + 1 = 19
Simple_Hash ("ET") = ((68 + 83) mod 97) + 1 = 55
Simple_Hash ("B1") = ((66 + 49) mod 97) + 1 = 19
Simple_Hash ("C0") = ((67 + 48) mod 97) + 1 = 19
```

INSTRUCTOR NOTES

DO NOT GO INTO DETAIL ABOUT HOW Division_Hash WORKS! JUST GET ACROSS THE IDEA THAT CERTAIN HASH FUNCTIONS ARE BETTER THAN OTHERS. THE INFORMATION BELOW IS ONLY BACKGROUND INFORMATION FOR YOU.

THIS HASHING FUNCTION IS CALLED THE DIVISION METHOD AND IS ONE OF THE MORE POPULAR HASHING FUNCTIONS, AND ONE OF THE EARLIEST. A GREAT DEAL OF RESEARCH HAS BEEN DONE ON HASHING FUNCTIONS, TO WHICH WE DIRECT YOU.

THE HASHING FUNCTION SIMPLY COMPUTES

```
            ordinal value of (Key) mod Table_Size
```

NOTICE THAT WE USE A PRIME NUMBER, 97, AS A TABLE SIZE. RESEARCH HAS SHOWN THAT PRIME NUMBERS ARE BEST, SINCE THEY RESULT IN A GOOD SCATTER. IF YOU DO NOT USE A PRIME YOU CAN GET SOME BAD RESULTS. FOR EXAMPLE, IF WE USE 65 FOR THE TABLE SIZE AND ALL THE KEYS ARE EQUAL mod 5 (A AND B ARE EQUAL mod 5 IF (A - B) mod 5 = 0). THEN THE KEYS WILL HASH TO AT MOST 13 DISTINCT VALUES.

NOTE AGAIN THAT WE HAVE CREATED A TYPE LARGE ENOUGH TO HOLD OUR COMPUTATIONS, AND MADE Table_Range_Type A SUBTYPE OF IT. WE COULD PERFORM THE ARITHMETIC IN type Positive, BUT IT IS GOOD PRACTICE NOT TO DO SO. BY GIVING THE ACTUAL RANGE WE NEED, WE CAN CATCH ANY UNEXPECTED OVERFLOW THAT MIGHT OCCUR DURING A COMPUTATION. WHILE THIS CANNOT HAPPEN IN THIS EXAMPLE, IT IS STILL A GOOD PRACTICE TO GET INTO.

FOR LARGE STRINGS, CALCULATING THE ORDINAL VALUE OF THE KEY WILL EXCEED THE ARITHMETIC CAPABILITIES OF THE PROCESSOR. THUS WE PERFORM THE mod AT EVERY ITERATION. THIS IS VALID SINCE (A + B) mod C = ((A mod C) + B) mod C. THIS IS THE TYPE OF "ADJUSTMENT" THAT MUST OFTEN BE MADE TO AN ALGORITHM IN ORDER TO USE IT ON A COMPUTER.

A BETTER HASHING FUNCTION FOR STRINGS

- THE HASH FUNCTION ON THE PREVIOUS SLIDE, WHICH SUMS ASCII VALUES IN A STRING, CAUSES ANAGRAMS (E.G. "STOP", "POTS", "TOPS", "SPOT") TO COLLIDE.

- THE FUNCTION BELOW ESSENTIALLY TREATS EACH CHARACTER OF THE STRING AS A DIGIT IN A BASE-128 NUMBER, AND RETURNS THE VALUE OF THAT NUMERAL MOD Table_Size:

```
function Division_Hash (Key : in String) return Table_Range_Type is

   Hash_Value : Hash_Range_Type := 0;
   Char_Value : Hash_Range_Type;
   Radix      : constant := 128;

begin -- Division_Hash

   -- COMPUTE ORDINAL VALUE (Key) mod Table_Size

   for I in Key'Range loop
      Char_Value := Character'Pos (Key (I));
      Hash_Value := (Hash_Value * Radix + Char_Value) mod Table_Size;
   end loop;
   return Hash_Value + 1;

end Division_Hash;
   ...
```

- COLLISIONS LESS LIKELY BECAUSE POSITION OF CHARACTERS IS SIGNIFICANT.

```
Division_Hash ("A2") = ((65 * 128 + 50) mod 97) + 1 = 29
Division_Hash ("B1") = ((66 * 128 + 49) mod 97) + 1 = 59
Division_Hash ("C0") = ((67 * 128 + 48) mod 97) + 1 = 89
```

INSTRUCTOR NOTES

HOW FREQUENTLY COLLISIONS OCCUR CAN DEPEND ON THE HASHING FUNCTION. THE SIMPLE HASHING FUNCTION WE PRESENTED MAPPED TOO MANY KEYS TO THE SAME VALUE, THEREBY INCREASING THE CHANCE OF COLLISIONS.

HOW COLLISIONS ARE HANDLED CAN ALSO AFFECT THE COLLISION RATE.

"Language" AND "System" COLLIDE WITH EACH OTHER.

"Army", "Navy", "DoD", and "Ironman" COLLIDE WITH EACH OTHER.

COLLISIONS

- LET H BE A HASHING FUNCTION ON STRINGS SUCH THAT:
  - "Ada" IS MAPPED TO 1
  - "Program" IS MAPPED TO 2
  - "Language" AND "System" ARE MAPPED TO 3
  - "Army", "Navy", "DoD", AND "Ironman" ARE MAPPED TO 9
- WHICH STRINGS COLLIDE?

INSTRUCTOR NOTES

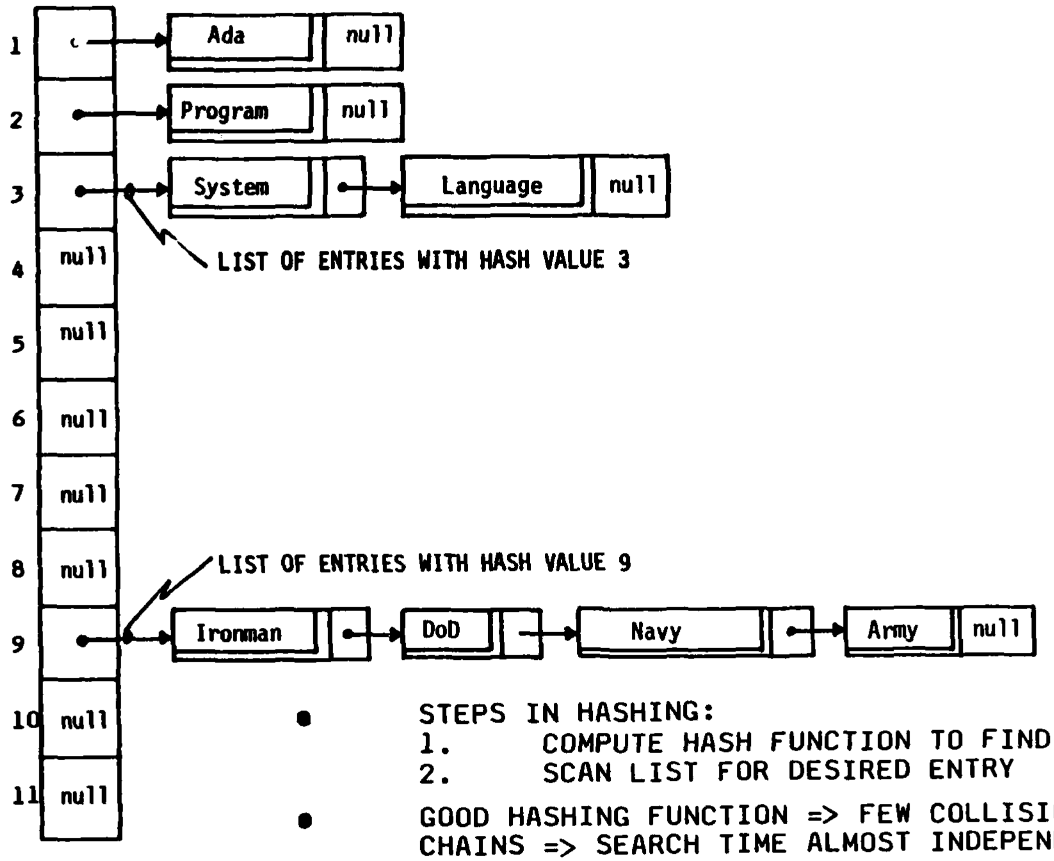
IN CHAINING, THE HASH TABLE IS ACTUALLY A TABLE OF LINKED LISTS -- CALLED BUCKETS. THE HASHING FUNCTION SIMPLY SELECTS ONE OF THE BUCKETS.

UNLESS A KEY HAS HASHED TO A GIVEN ENTRY, THE LIST IS EMPTY. IF A KEY HASHES TO THAT ENTRY IT BECOMES THE FIRST ITEM IN THE LIST. IF ANOTHER KEY HASHES TO THAT SAME VALUE, I.E., IF A COLLISION OCCURS, THEN IT IS JUST DROPPED INTO THE SAME BUCKET. THE NEW ENTRY IS ADDED TO THE FRONT OF THE LIST.

NOW NOTICE THAT THE SIZE OF THE TABLE DOES NOT LIMIT THE NUMBER OF ENTRIES. THE MAXIMUM NUMBER OF PROBES IS THE LENGTH OF THE LONGEST LIST.

THE NUMBER OF COLLISIONS DEPENDS ON THE HASHING FUNCTION ONLY.

CHAINING -- EXAMPLE



- STEPS IN HASHING:
  1. COMPUTE HASH FUNCTION TO FIND DESIRED LIST
  2. SCAN LIST FOR DESIRED ENTRY
- GOOD HASHING FUNCTION => FEW COLLISIONS => SHORT CHAINS => SEARCH TIME ALMOST INDEPENDENT OF NUMBER OF ENTRIES
- BAD HASHING FUNCTION => MANY COLLISIONS => LONG CHAINS => SEARCH TIME PROPORTIONAL TO NUMBER OF ENTRIES (LINEAR SEARCH)

INSTRUCTOR NOTES

WE ONLY DISCUSS THE PACKAGE SPECIFICATION.

THIS DIFFERS FROM THE LINEAR SEARCH IN THAT

- THE TABLE SIZE IS FOR THE NUMBER OF BUCKETS, NOT THE NUMBER OF ENTRIES
- A HASHING FUNCTION IS REQUIRED.

THIS DIFFERS FROM THE SEARCH TREE IN THAT

- A TABLE SIZE IS REQUIRED FOR THE NUMBER OF BUCKETS
- A HASHING FUNCTION IS REQUIRED
- A Matching_Keys FUNCTION RATHER THAN AN ORDERING FUNCTION IS NEEDED.

IT DIFFERS FROM BOTH IN THAT WE ARE PROVIDING AN ABSTRACT TYPE Hash_Table. THUS MANY HASH TABLE OBJECTS MAY EXIST, SO THE PROCEDURES Update_Data AND Look_Up_Data REQUIRE A PARAMETER OF TYPE Hash_Table_Type. SINCE THIS IS ALL WE NEED TO BE ABLE TO DO WITH A HASH TABLE OUTSIDE OF THE PACKAGE, WE MAKE Hash_Table_Type LIMITED PRIVATE. (WERE THE TYPE NOT LIMITED, A USER COULD ASSIGN HASH TABLES. THIS WOULD LEAD TO SHARING OF BUCKETS, SO THAT AN OPERATION ON ONE HASH TABLE COULD HAVE A SIDE-EFFECT ON ANOTHER.)

THE HASH TABLE SIZE DETERMINES NOT THE NUMBER OF ENTRIES THAT CAN BE STORED, BUT THE NUMBER OF LISTS OVER WHICH THOSE ENTRIES WILL BE DISTRIBUTED. IT IS A PERFORMANCE PARAMETER RATHER THAN A LOGICAL CAPACITY. SINCE A LARGER TABLE USUALLY MEANS FEWER COLLISIONS, THE CHOICE OF TABLE SIZE ENTAILS A TIME/SPACE TRADEOFF.

NOTICE THAT THE HASHING FUNCTION RETURNS POSITIVE VALUES. THIS ALLOWS US TO SPECIFY DIFFERENT TABLE SIZES WITHOUT CHANGING THE HASHING FUNCTION.

HASH TABLE PACKAGE SPECIFICATION

```
with List_Package_Template;      -- USE AN ALREADY-EXISTING IMPLEMENTION OF LINKED LISTS
generic
   type Key_Type is private;
   type Data_Type is private;
   Null_Data  : in Data_Type;
   Table_Size : in Integer := 97;      -- 97 IS PRIME
   with Hash_Value (Key : in Key_Type) return Positive;
package Lookup_Table_Package is
   type Hash_Table_Type is limited private; -- THIS PACKAGE PROVIDES A TYPE FOR LOOKUP TABLES
                                            -- RATHER THAN OPERATIONS ON A SINGLE TABLE.
   procedure Update_Data
      (Hash_Table: in out Hash_Table_Type; Key: in Key_Type; Data : in Data_Type);
   procedure Look_Up_Data
      (Hash_Table: in Hash_Table_Type; Key: in Key_Type; Data : out Data_Type);
private
   type Table_Entry_Type is
      record
         Key_Part : Key_Type;
         Data_Part : Data_Type;
      end record;
   package Bucket_Package is new List_Package_Template (Table_Entry_Type);
   subtype Bucket_Type is Bucket_Package.List_Type;
   subtype Hash_Table_Range is Positive range 1 .. Table_Size;
   type Hash_Table_Type is array (Hash_Table_Range) of Bucket_Type;
end Lookup_Table_Package;
```

INSTRUCTOR NOTES

WE CREATE AN INSTANCE OF THE Hash_Table_Package CALLED Alphabetic_Hash_Table. THE Key_Type IS A SUBTYPE OF String AND THE HASHING FUNCTION WE USE IS THE Division_Hash FOR STRINGS THAT WE USED EARLIER.

AN EXAMPLE OF A HASH TABLE DECLARATION

```
subtype Word_Subtype is String (1 .. 11);
type Definition_Type is ---;
Null_Definition: constant Definition_Type := ---;
   .
   .
   .
package Alphabetic_Hash_Package is
   new Lookup_Table_Package
      (Key_Type   =>  Word_Subtype,
       Data_Type  =>  Definition_Type,
       Null_Data  =>  Null_Definition,
       Hash_Value =>  Division_Hash);
Alphabetic_Hash_Table : Alphabetic_Hash_Package.Hash_Table_Type;
   .
   .
   .
   Defined_As : Definition_Type;
   .
   .
   .
Alphabetic_Hash_Package.Look_Up_Data (Alphabetic_Hash_Table, "ABSTRACTION", Defined_As);
   .
   .
   .
```

INSTRUCTOR NOTES

THIS TYPE OF SEARCHING OCCURS FREQUENTLY. THE IMPLEMENTATION TECHNIQUE WE USE DEPENDS ON THE PERFORMANCE WE NEED.

PRIORITY QUEUES

- COLLECTIONS OF ITEMS WITH "PRIORITIES"
- TWO OPERATIONS
  - -- ADD ITEM TO QUEUE
  - -- EXTRACT ITEM WITH HIGHEST PRIORITY
- EXAMPLES OF PRIORITIES:
  - -- EXECUTION TIME LIMIT FOR A JOB QUEUE (SHORTEST TIME = HIGHEST PRIORITY)
  - -- ALPHABETICAL ORDER FOR A DICTIONARY (LOWEST STRING = HIGHEST PRIORITY)
  - -- INTERRUPT LEVEL (HIGHEST LEVEL = HIGHEST PRIORITY)

INSTRUCTOR NOTES

THE LINKED LIST VERSION WILL USE THE LINKED LIST DEVELOPED AS A LAB EXERCISE.

WE WILL INTRODUCE THE HEAP LATER.

THE DETAILS OF THE HEAP IMPLEMENTATION WILL NOT BE GIVEN.

TWO PRIORITY QUEUE IMPLEMENTATIONS

- LINKED LIST VERSION
- HEAP VERSION

INSTRUCTOR NOTES

THIS EXAMPLE SUGGESTS HOW A GENERIC PRIORITY QUEUE PACKAGE MIGHT BE USED. THE SECOND GENERIC PARAMETER GIVES THE CRITERION FOR CHOOSING WHICH ENTRY TO EXTRACT FIRST. IN THIS CASE THE FUNCTION Has_Shorter_Execution_Time_Limit PROVIDES THE CRITERION. A FULL GENERIC SPECIFICATION IS GIVEN ON THE NEXT SLIDE.

THIS EXAMPLE IMPLEMENTS JOB QUEUE BASED ON SHORTEST EXECUTION TIME LIMIT. THE JOB WITH THE SMALLEST EXECUTION TIME LIMIT HAS THE HIGHEST PRIORITY.

HOW WE MIGHT USE A GENERIC PRIORITY QUEUE PACKAGE

```
with Calendar; use Calendar;

type Job_Type is ---;
type Job_Entry_Type is
   record
      Execution_Time_Limit : Duration;
      Job_Part : Job_Type;
   end record;

Job : Job_Entry_Type;

function Has_Shorter_Execution_Time_Limit (Job_1, Job_2 : Job_Entry_Type)
   return Boolean is
begin -- Has_Shorter_Execution_Time_Limit
   return Job_1.Execution_Time_Limit < Job_2.Execution_Time_Limit;
end Has_Shorter_Execution_Time_Limit;

package Job_Queue_Package is
   new Priority_Queue_Package
      (Element_Type             =>  Job_Entry_Type,
       Has_Higher_Priority_Than =>  Has_Shorter_Execution_Time_Limit);

Job_Queue : Job_Queue_Package.Queue_Type;
Job       : Job_Type;
...
Job_Queue_Package.Add_Element (Job_Queue, Job);
...
if not Job_Queue_Package.Empty (Job_Queue) then
   Job_Queue_Package.Extract_Element (Job_Queue, Job);
   Begin_Job (Job);
end if;
```

INSTRUCTOR NOTES

THE INTERFACE VARIES GREATLY IN THIS PACKAGE.

- A PRIORITY FUNCTION IS REQUIRED. NOTE THAT THIS IS SIMILAR TO AN ORDERING FUNCTION.

- OUR OPERATIONS ARE ADDING AND REMOVING RATHER THAN UPDATING AND LOOKING UP. WE ALSO NEED TO KNOW IF THE QUEUE IS EMPTY AND IF WE ATTEMPT TO EXTRACT FROM AN EMPTY QUEUE, WE RAISE AN EXCEPTION.

- WE PROVIDE AN ABSTRACT TYPE -- Queue_Type -- WHICH WE PROVIDE ALL NECESSARY OPERATIONS ON. SINCE WE DO NOT WANT TO ALLOW ASSIGNMENT OR EQUALITY OPERATIONS, WE MAKE THE TYPE LIMITED.

THE DERIVED TYPE DECLARATION FOR Queue_Type IS REQUIRED BECAUSE THE PRIVATE TYPE DECLARATION IN THE VISIBLE PART MUST BE MATCHED BY A TYPE DECLARATION IN THE PRIVATE PART. THE SUBTYPE DECLARATION

```
            subtype Queue_Type is Queue_Package.List_Type;
```

WOULD NOT DO.

STUDENTS WILL HAVE AN EXERCISE TO MAINTAIN THE QUEUE IN ORDER, SO EXPLAIN THIS IN DETAIL.

Priority_Queue_Package SPECIFICATION -- LINKED LIST VERSION

```
with List_Package_Template;
generic

   type Element_Type is private;

   with function Has_Higher_Priority_Than
      (Element_1, Element_2: Element_Type) return Boolean;

package Priority_Queue_Package is

   type Queue_Type is limited private;

   procedure Add_Element (Queue : in out Queue_Type; Element : in Element_Type);

   procedure Extract_Element (Queue : in out Queue_Type; Element : out Element_Type);

   function Empty (Queue : Queue_Type) return Boolean;

   Empty_Queue_Error : exception;  -- RAISED BY Extract_Element WHEN CALLED WITH AN EMPTY QUEUE

private

   package Queue_Package is new
      List_Package_Template (Element_Type);

   type Queue_Type is new Queue_Package.List_Type; -- DERIVED TYPE DECLARATION SERVING AS THE
                                                   -- FULL DECLARATION OF A LIMITED PRIVATE TYPE
end Priority_Queue_Package;
```

INSTRUCTOR NOTES

WE USE THE LINKED LIST PACKAGE YOU DEVELOPED AS AN EXERCISE.

NOTE THAT WE OVERLOADED "<" TO USE FOR ORDERING ELEMENTS.

Priority_Queues PACKAGE BODY -- LINKED LIST VERSION

```
package body Priority_Queue_Package is

   subtype Position_Type is Queue_Package.Position_Type;

   function Element_Value (Position : Position_Type) return Element_Type
      renames Queue_Package.Element_Value;

   function Next_Position (Position : Position_Type) return Position_Type
      renames Queue_Package.Next_Position;

   Null_Position : constant Position_Type
      renames Queue_Package.Null_Position;

   -- THE FOLLOWING SUBPROGRAMS ARE INHERITED FROM THE PARENT TYPE:
    -- function First_Position (Queue : Queue_Type) return Position_Type;

   -- procedure Delete_Element (Queue : in out Queue_Type; Position : Position_Type);

   -- procedure Insert_Element (Queue   : in out Queue_Type;
                                Element : in Element_Type;
                                After   : in Position_Type);

   -- function Length (Queue : Queue_Type) return Natural;

   -- CONTINUED ON NEXT SLIDE
```

INSTRUCTOR NOTES

ADDING AN ELEMENT SIMPLY INVOLVES ADDING AN ELEMENT TO THE LIST. NOTE THAT WE TAKE ADVANTAGE OF After => Null_Position.

PRIORITY QUEUES LINKED LIST VERSION -- Empty AND Add_Element

```
function Empty (Queue : Queue_Type) return Boolean is
begin -- Empty
   return Length (Queue) = 0;
end Empty;

procedure Add_Element (Queue : in out Queue_Type; Element : in Element_Type) is
begin -- Add_Element

   Insert_Element (Queue, Element, Null_Position);  -- INSERT AT FRONT OF LIST

end Add_Element;

-- CONTINUED ON NEXT SLIDE
```

INSTRUCTOR NOTES

EXTRACTING CAUSES THE LARGEST ELEMENT TO BE RETURNED AND REMOVED FROM THE LIST.

THIS IS SIMPLY A LINEAR SEARCH THROUGH A LINKED LIST WITH n/2 AVERAGE COMPARISONS.

PRIORITY QUEUES LINKED LIST VERSION -- Extract_Element

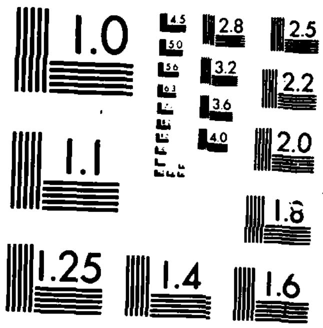
```
procedure Extract_Element (Queue : in out Queue_Type; Highest : out Element_Type) is
begin -- Extract_Element
   if Length (Queue) = 0 then
      raise Empty_Queue_Error;
   else
      declare
         Position_Of_Highest : Position_Type := First_Position (Queue);
         Highest_So_Far      : Element_Type := Element_Value (Position_Of_Highest);
         Position            : Position_Type := Next_Position (Position_Of_Highest);
         Element             : Element_Type;
      begin
         while Position /= Null_Position loop
            Element := Element_Value (Position);
            if Has_Higher_Priority_Than (Element, Highest_So_Far) then
               Position_Of_Highest := Position;
               Highest_So_Far := Element;
            end if;
            Position := Next_Position (Position);
         end loop;

         Delete_Element (Queue, Position_Of_Highest);
         Highest := Highest_So_Far;

      end;

   end if;

end Extract_Element;

end Priority_Queue_Package;
```

INSTRUCTOR NOTES

THE HEAP WILL PROVIDE US WITH A RADICALLY DIFFERENT IMPLEMENTATION OF PRIORITY QUEUES.

AD-A165 076 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506 6/7

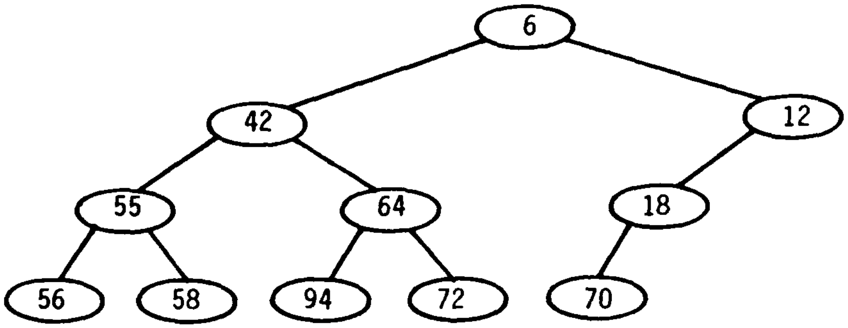
UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

HEAP

- A HEAP IS A TREE SUCH THAT
  - EVERY LEVEL OF THE TREE IS FULL, EXCEPT THAT NODES MAY BE MISSING FROM THE RIGHT END OF THE BOTTOM LEVEL.
  - THE PRIORITY OF ANY NODE IS GREATER THAN OR EQUAL TO THE PRIORITY OF ITS CHILDREN.



- IN THIS EXAMPLE, THE LOWER VALUE IS THE HIGHER PRIORITY (POSSIBLY EXECUTION TIME LIMIT). AS WE FOLLOW ANY PATH FROM THE ROOT TO A LEAF, WE ENCOUNTER NODES WITH LOWER AND LOWER PRIORITIES.

- THE DEFINITION OF A HEAP IMPLIES THAT THE ROOT OF THE TREE CONTAINS THE ELEMENT OF HIGHEST PRIORITY.

INSTRUCTOR NOTES

THIS IS AN EXAMPLE OF REMOVING THE HIGHEST PRIORITY ELEMENT FROM A HEAP AND THEN RESTORING THE TREE TO A HEAP. SINCE THE SMALLEST VALUE IS THE HIGHEST PRIORITY IN THIS EXAMPLE, WE WILL DEAL WITH VALUES RATHER THAN PRIORITY.

ONCE WE REMOVE THE ROOT VALUE WE REPLACE IT WITH A LEAF. WE CHOOSE TO ALWAYS SELECT THE LONGEST, RIGHTMOST PATH. THIS WILL HELP KEEP THE TREE BALANCED.

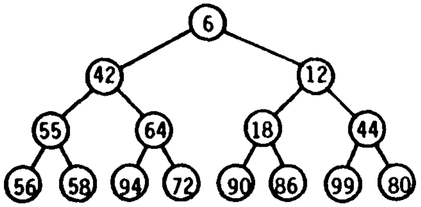
THE RESULT OF MOVING THE LEAF IS SHOWN IN THE SECOND TREE. SINCE 80 IS SMALLER THAN 42 AND 12, THE TREE IS NOT A HEAP. IN ORDER TO RECTIFY THIS, WE MUST MOVE 80 LOWER IN THE TREE.

WE SWAP 80 AND 12 TO GET THE THIRD TREE. THE ROOT IS NOW SMALLER THAN ITS CHILDREN, BUT THE TREE IS STILL NOT A HEAP, SINCE THE SUBTREE ROOTED AT 80 IS NOT A HEAP. NOTE THAT IF WE HAD SWAPPED 42 AND 80, THE ROOT WOULD NOT HAVE SATISFIED THE HEAP CONDITION; THUS WE MUST ALWAYS SWAP WITH THE SMALLER CHILD.

THE FOURTH TREE SHOWS THE RESULT OF APPLYING THE ABOVE TO THE SUBTREE ROOTED AT 80. WE FINALLY HAVE A HEAP.

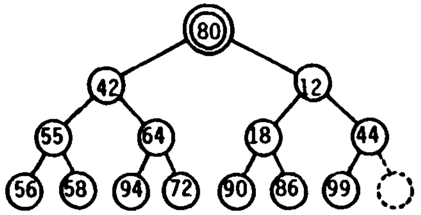
REMOVING THE HIGHEST PRIORITY ELEMENT

(IN THIS EXAMPLE, SMALLEST VALUE IS HIGHEST PRIORITY)



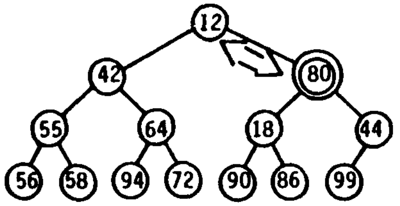
EXTRACT THE VALUE AT THE ROOT.
SMALLEST VALUE IS 6

(NOT A HEAP)

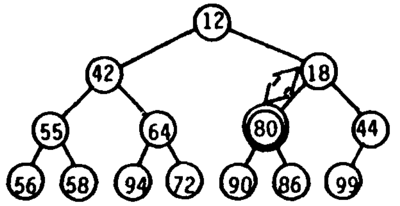


COPY THE VALUE IN THE RIGHT MOST LEAF ON THE BOTTOM LEVEL TO THE ROOT, THEN REMOVE THE LEAF

(NOT A HEAP)



(THIS IS A HEAP)

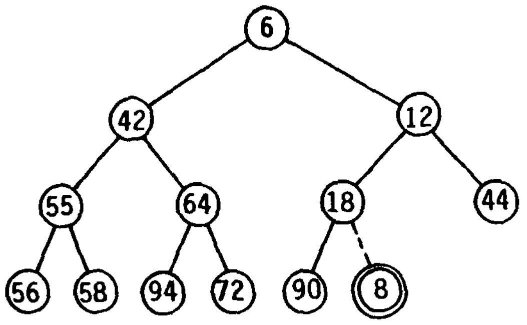


WHILE THE COPIED VALUE WAS A HIGHER-PRIORITY CHILD loop
SWAP THE VALUE WITH ITS HIGHEST-PRIORITY CHILD; end loop;

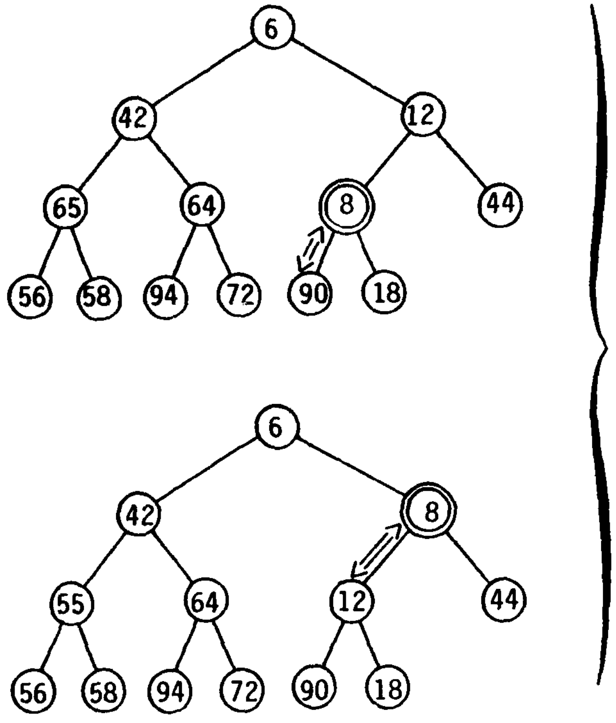
INSTRUCTOR NOTES

IN THIS CASE, WE JUST "BUBBLE" THE NEW VALUE UP THE TREE TO ITS RIGHTFUL POSITION.

INSERTING A NEW ELEMENT



ADD A NEW LEAF AT THE LEFT MOST OPEN POSITION IN THE BOTTOM ROW. (START A NEW ROW IF THE BOTTOM ROW IS FULL.)



```
while THE NEWLY ADDED VALUE HAS
HIGHER PRIORITY THAN ITS PARENT
loop SWAP THE VALUE WITH ITS
PARENT;
end loop;
```

INSTRUCTOR NOTES

IN THE TREE DRAWING, NUMBERS IN THE CIRCLES ARE DATA AND NUMBERS OUTSIDE THE CIRCLES ARE NODE NUMBERS.

IN THE ARRAY DRAWING, QUESTION MARKS REPRESENT ARBITRARY VALUES. EXTRA ARRAY COMPONENTS ARE PRESENT TO ALLOW THE HEAP TO GROW. WE MUST KEEP TRACK OF THE CURRENT SIZE OF THE HEAP (13 IN THIS CASE) TO KNOW WHICH ARRAY COMPONENTS CONTAIN MEANINGFUL VALUES.

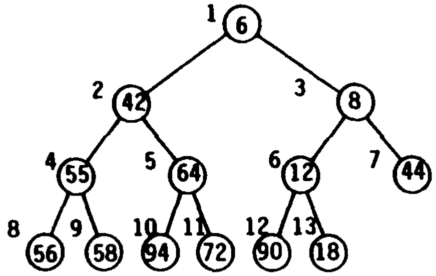
THE TYPE DECLARATION IS OF THE FORM SEEN IN SECTION 4 FOR A VARIABLE-LENGTH LIST WITH A MAXIMUM LENGTH GIVEN BY A DISCRIMINANT.

THE NEXT SLIDE SHOWS HOW HEAP <u>OPERATIONS</u> ARE IMPLEMENTED.

REPRESENTING THE HEAP

- THE TREE GROWS ROW BY ROW, FILLING IN THE BOTTOM ROW LEFT TO RIGHT ONCE ALL ROWS ABOVE IT ARE FULL.

- TREE CAN THEREFORE BE REPRESENTED AS AN ARRAY, AS DESCRIBED IN SECTION 15:
  - NUMBER NODES Row_By_Row, LEFT TO RIGHT IN EACH Row.
  - ARRAY ELEMENT n CONTAINS DATA FOR Row NUMBERED n:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
6	42	8	55	64	12	44	56	58	94	72	90	18	?	?	?	?	?

Current size = 13

- type DECLARATIONS:

```
type Node_List_Type is array (Positive range <>) of Data_Type;

type Heap_Type (Maximum_Size : Natural) is
   record
      Current_Size_Part : Natural = 0;
      Node_List_Part    : Node_List_Type (1 .. Maximum_Size);
   end record;
```

ANSWERS:

- CONTENTS OF HEAP H's ROOT: H.Node_List_Part (1)
- NODE NUMBER OF NODE N's LEFT CHILD: 2*N

  (NODE NUMBER N HAS NO CHILDREN IF 2*N>H.Current_Size_Part.)
- NODE NUMBER OF NODE N's RIGHT CHILD: 2*N+1

  (NODE N HAS ONLY A LEFT CHILD IF 2*N = H.Current_Size_Part.)
- NODE NUMBER OF NODE N's PARENT: N/2 -- INTEGER DIVISION
- VALUE IN RIGHTMOST LEAF ON BOTTOM LEVEL:

  H.Node_List_Part (H.Current_Size_Part)
- REMOVING RIGHTMOST LEAF ON BOTTOM LEVEL:

  H.Current_Size_Part := H.Current_Size_Part - 1;

THE ONLY CUMBERSOME OPERATION IS CHECKING WHETHER A NODE HAS A HIGHER PRIORITY CHILD AND, IF SO, SWAPPING CONTENTS WITH THAT CHILD. THIS OPERATION IS STRAIGHTFORWARD, BUT MESSY BECAUSE OF THE POSSIBILITY THAT A NODE WILL HAVE FEWER THAN TWO CHILDREN.

THE OPERATIONS SHOWN ARE ONLY MEANINGFUL WHEN CERTAIN OBVIOUS ASSUMPTIONS APPLY. IT ONLY MAKES SENSE TO TALK ABOUT A PARTICULAR NODE (THE ROOT, THE LAST NODE, NODE N) WHEN THE HEAP IS NONEMPTY (I.E., H.Current_Size_Part > 0); A LEAF CAN ONLY BE ADDED TO A HEAP THAT IS NOT FULL TO ITS DECLARED CAPACITY (I.E., H.Current_Size_Part < H.Maximum_Size); NODE 1 (THE ROOT) HAS NO PARENT.

MANIPULATING THE HEAP

- GIVEN THIS REPRESENTATION, IT IS EASY TO IMPLEMENT THE PRIMITIVE TREE OPERATIONS NEEDED TO EXTRACT OR INSERT AN ITEM:

  - CONTENTS OF HEAP H's NODE NUMBER N: H.Node_List_Part (N)
  - CONTENTS OF HEAP H's ROOT: □
  - NODE NUMBER OF NODE N's LEFT CHILD: □

    (NODE NUMBER N HAS NO CHILDREN IF □.)
  - NODE NUMBER OF NODE N's RIGHT CHILD: □

    (NODE N HAS ONLY A LEFT CHILD IF □.)
  - NODE NUMBER OF NODE N's PARENT: □
  - ADDING VALUE X AT THE LEFTMOST OPEN POSITION IN BOTTOM ROW OF HEAP H (STARTING A NEW ROW IF THE BOTTOM ROW IS FULL):

    ```
    H.Current_Size_Part := H.Current_Size_Part + 1;
    H.Node_List_Part (H.Current_Size_Part) := X;
    ```
  - VALUE IN RIGHTMOST LEAF ON BOTTOM LEVEL: □
  - REMOVING RIGHTMOST LEAF ON BOTTOM LEVEL: □

INSTRUCTOR NOTES

HEAP APPEARS TO BE BETTER.

PICKING AN "IMPROPER" IMPLEMENTATION CAN HAVE A BAD EFFECT ON PERFORMANCE.

COMPARING THE TWO IMPLEMENTATIONS OF PRIORITY QUEUES

- DISTINCTLY DIFFERENT IMPLEMENTATIONS
- BOTH PROVIDE FOR DYNAMIC GROWTH
- LINKED LIST IMPLEMENTATION (STRAIGHTFORWARD)
  - -- EXTRACTION
    - -- AVERAGE PERFORMANCE ORDER (n/2)
    - -- WORST PERFORMANCE ORDER (n)
  - -- INSERTION
    - -- CONSTANT
- HEAP IMPLEMENTATION (INTRICATE)
  - -- EXTRACTION
    - -- AVERAGE PERFORMANCE ORDER ($\log_2 n$)
    - -- WORST PERFORMANCE ORDER ($\log_2 n$)
  - -- INSERTION
    - -- ORDER ($\log_2 n$)
  - -- NOTE $\log_2$ IS THE DEPTH OF AN n-NODE HEAP

INSTRUCTOR NOTES

SECTION 17

SORTING

INSTRUCTOR NOTES

FOR QUICKSORT, WE WILL LOOK AT A RECURSIVE AND NON-RECURSIVE IMPLEMENTATION.

FOR THE HEAP SORT, WE'LL LOOK AT HOW A HEAP CAN BE USED AND GIVEN SOME EXAMPLES, BUT WE WILL NOT PRESENT AN ACTUAL IMPLEMENTATION.

SORTING -- TOPICS

- QUICKSORT
- HEAP SORT

FURTHER DETAILS IN EXERCISE 5.2 OF ADVANCED Ada WORKBOOK.

INSTRUCTOR NOTES

WE WILL SEE HOW RARELY THE WORST CASE OCCURS.

WE WILL SEE HOW A SIMPLE DECISION IN THE IMPLEMENTATION CAN AFFECT THE WORST CASE FREQUENCY.

QUICKSORT

- BEST IN-MEMORY SORT
- AVERAGE PERFORMANCE ORDER (n $\log_2 n$)
- WORST CASE PERFORMANCE ORDER (n^2)
  - -- OCCURS RARELY

INSTRUCTOR NOTES

QUICKSORT IS A GOOD ILLUSTRATION OF RECURSION. TO SORT THE LEFT AND RIGHT PARTITIONS WE RECURSIVELY INVOKE QUICKSORT.

SINCE WE HAVE FOR ANY $x \in$ LEFT PARTITION AND ANY $y \in$ RIGHT PARTITION

$$x \leq \text{splitting element} \leq y$$

IF WE HAD THE LEFT PARTITION SORTED AND THE RIGHT PARTITION SORTED, THEN THE ARRAY WOULD BE SORTED.

QUICKSORT -- HOW IT WORKS

- DIVIDES AN ARRAY INTO
  - -- LEFT PARTITION
  - -- SPLITTING ELEMENT
  - -- RIGHT PARTITION
- THE FOLLOWING RELATIONSHIPS HOLD:
  - -- THE SPLITTING ELEMENT IS GREATER THAN OR EQUAL TO ANY ELEMENT IN THE LEFT PARTITION
  - -- THE SPLITTING ELEMENT IS LESS THAN OR EQUAL TO ANY ELEMENT IN THE RIGHT PARTITION
  - -- CONSEQUENTLY, THE SPLITTING ELEMENT IS IN ITS CORRECT FINAL POSITION AND ALL OTHER ELEMENTS ARE ON THE CORRECT SIDE OF THE SPLITTING ELEMENT
- APPLIES QUICKSORT RECURSIVELY TO THE LEFT PARTITION AND THE RIGHT PARTITION

INSTRUCTOR NOTES

QUICKSORT DETERMINES THREE COMPONENTS: A SPLITTING ELEMENT (SHOWN AS A CIRCLE), A LEFT PARTITION, AND A RIGHT PARTITION.

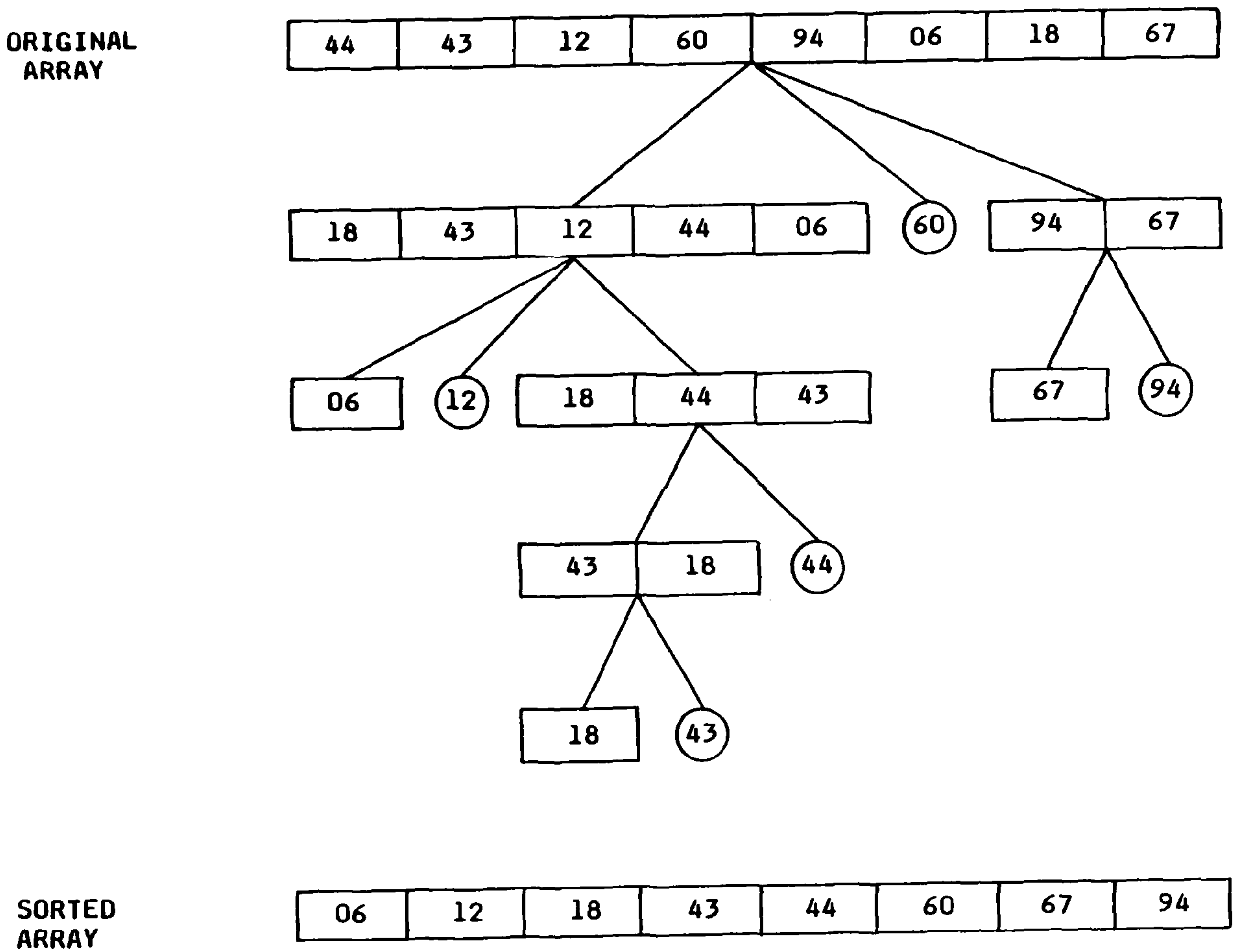
NOTE THAT THE LEAVES IN THIS "TREE," READ LEFT-TO-RIGHT, ARE THE SORTED ARRAY.

THIS FIGURE SHOWS THE DEPTH OF RECURSION NEEDED.

NOTE THAT IF WE ARE CLEVER ENOUGH TO AVOID DEALING WITH THE TRIVIAL CASES WE CAN ELIMINATE ONE LEVEL OF RECURSION.

WE WILL LOOK AT THE SPECIFICATION SEPARATELY.

QUICKSORT EXAMPLE -- INTEGER ARRAY



INSTRUCTOR NOTES

WE IMPLEMENT QUICKSORT AS A GENERIC PROCEDURE AND ALLOW THE ARRAY TO BE PASSED AS AN ARGUMENT.

ANSWERS:

```
   1.  package Sort_Integers_Ascending is
          new Quick_Sort (Element_Type => Integer, Table_Type => Integer_List_Type);

2.  package Sort_Floating_Numbers_Descending is
       new Quick_Sort (Element_Type => Float, Table_Type => Float_List_Type, "<" => ">");
```

QUICKSORT -- SPECIFICATION

```
generic

   type Element_Type is private;
   type Table_Type is array (Positive range <>) of Element_Type;

   with function "<" (Left, Right : Element_Type) return Boolean is "<>";

procedure Quick_Sort (Table : in out Table_Type);
```

- EXERCISE: ASSUMING THE DECLARATIONS

```
type Integer_List_Type is array (Positive range<>) of Integer;
type Float_LIst_Type is array (Positive range<>) of Float;
```

1. INSTANTIATE Quick_Sort TO SORT AN Integer_List_Type ARRAY IN ASCENDING ORDER.

2. INSTANTIATE Quick_Sort TO SORT A Float_List_Type ARRAY IN DESCENDING ORDER.

INSTRUCTOR NOTES

WE SHOW THE IMPLEMENTATION OF THIS PROCEDURE BUT NOT OF Split.

THIS EXAMPLE ILLUSTRATES THE PARTITION OF QUICKSORT. THE ACTUAL CREATION OF PARTITIONS IS ACCOMPLISHED IN THE SEPARATELY COMPILED PROCEDURE Split.

NOTICE THE USE OF ARRAY SLICES. AN ALTERNATE APPROACH WOULD HAVE BEEN TO PASS THE ARRAY BOUNDS, BUT SLICES SEEM MORE NATURAL.

NOTE THE USE OF RENAMING DECLARATIONS TO ALLOW SPECIFICATION OF THE PROGRAM IN TERMS VERY CLOSE TO THE ENGLISH-LANGUAGE DESCRIPTION GIVEN EARLIER.

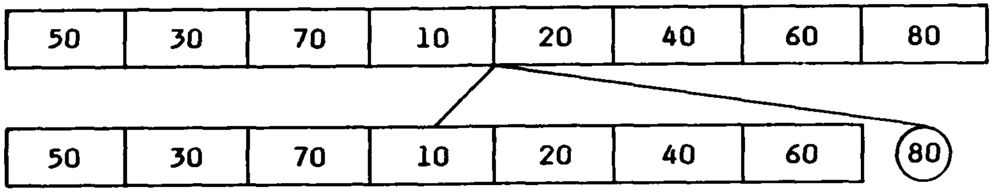
Quick_Sort PROCEDURE BODY

```
procedure Quick_Sort (Table : in out Table_Type) is

   procedure Split
      (A : in out Table_Type; Split_Position : out Positive)
      is separate;   -- NOT SHOWN HERE.  SEE ADVANCED Ada WORKBOOK EXERCISE 5.2

   procedure Sort (A : in out Table_Type) is

      Splitting_Position : Positive;

   begin -- Sort

      Split (A, Splitting_Position);

      declare  -- BLOCK STATEMENT
         Left_Partition : Table_Type renames A (A'First .. Splitting_Position - 1);
         Right_Partition : Table_Type renames A (Splitting_Position + 1 .. A'Last);
      begin
         if Left_Partition'Length > 1 then
            Sort (Left_Partition);
         end if;
         if Right_Partition'Length > 1 then
            Sort (Right_Partition);
         end if;
      end;  -- BLOCK STATEMENT

   end Sort;

begin -- Quick_Sort

   if Table'Length > 1 then
      Sort (Table);
   end if;

end Quick_Sort;
```

INSTRUCTOR NOTES

QUICKSORT'S WORST CASE

- WORST CASE IS ORDER (n^2)
- WORST CASE IS RARE
- OCCURS WHEN THE SPLITTING VALUE IS ALWAYS THE SMALLEST OR ALWAYS THE LARGEST ELEMENT IN THE ARRAY.



- IN THE WORST CASE, THE SPLIT ALWAYS YIELDS AN EMPTY PARTITION AND A PARTITION CONTAINING n-1 ELEMENTS.
- IF WE ALWAYS USE THE FIRST ELEMENT AS THE SPLITTING VALUE, THE WORST CASE WILL ARISE PRECISELY WHEN THE ARRAY IS ALREADY SORTED IN ASCENDING OR DESCENDING ORDER! SUCH CASES ARE NOT SO RARE.
- IF WE ALWAYS USE THE MIDDLE ELEMENT AS THE SPLITTING VALUE, A SORTED ARRAY WILL PRODUCE THE OPTIMAL SPLIT (50-50).

INSTRUCTOR NOTES

HEAP SORT USES THE PRIORITY QUEUE WE DESCRIBED EARLIER. WE WILL SEE THAT THIS CAN BE MADE MORE EFFICIENT.

THE PRIORITY QUEUE IS IMPLEMENTED AS A HEAP SO THAT INSERTION AND EXTRACTION ARE BOTH ORDER (log n).

HEAP SORT

- USES A PRIORITY QUEUE (IMPLEMENTED AS A HEAP)
- GOOD SORT TECHNIQUE
- ORDER ($n\log_2 n$) EVEN IN WORST CASE
- TYPICALLY ONLY 1/3 AS FAST AS QUICKSORT,
  BUT WORST CASE IS NOT AS BAD

INSTRUCTOR NOTES

THE Priority_Queue_Package WAS IMPLEMENTED USING A HEAP. WE USE THIS IN THE EXAMPLE:

- LOWER VALUES (ACCORDING TO THE GENERIC PARAMETER "<") CORRESPOND TO HIGHER PRIORITIES.

NOTE THAT THIS IS AN EFFECTIVE WAY TO SORT ON ARRAY; HOWEVER, IT IS COSTLY BECAUSE OF THE SPACE REQUIRED AND THE COPYING TO AND FROM THE ARRAY.

WE WILL LOOK AT SOMETHING MORE EFFECTIVE.

SORTING WITH THE PRIORITY QUEUE PACKAGE

```
generic
   type Element_Type is private;
   type Table_Type is array (Positive range < >) of Element_Type;
   with function "<" (Left, Right : Element_Type) return Boolean is < >;
procedure Heap_Sort (Table : in out Table_Type);

with Priority_Queue_Package;
procedure Heap_Sort (Table: in out Table_Type) is

   package Heap_Package is new
      Priority_Queue_Package  (Element_Type, Has_Higher_Priority_Than =>  "<");
      -- EXTRACT SMALLEST ELEMENTS FIRST

   Heap : Heap_Package.Queue_Type;

begin -- Heap_Sort

   -- BUILD A HEAP FROM THE TABLE:

   for I in Table'Range loop
      Heap_Package.Add_Element (Heap, Table (I));
   end loop;

   -- REFILL TABLE FROM HEAP, LOWER VALUES FIRST:

   for I in Table'Range loop
      Heap_Package.Extract_Element (Heap, Table (I));
   end loop;

end Heap_Sort;
```

INSTRUCTOR NOTES

THIS ALGORITHM COULD NOT HAVE BEEN DEVISED IF WE HAD VIEWED Queue_Type SIMPLY AS A PRIVATE TYPE. WE ARE EXPLOITING THE INTERNAL REPRESENTATION OF A PRIORITY QUEUE. OF COURSE WE CANNOT DO THIS USING Priority_Queue_Package, WHICH HIDES THIS INFORMATION FROM ITS USERS. WE MUST REÏMPLEMENT THE PRIORITY QUEUE FROM SCRATCH, GUARANTEEING THAT IT HAS THE REPRESENTATION WE ARE DEPENDING ON.

EXPANDING THE HEAP OPERATIONS IN LINE LEADS TO FURTHER SIMPLIFICATIONS. FOR EXAMPLE, THE FIRST STEP IN INSERTING A VALUE IN THE HEAP IS TO ADD A NEW NODE AT THE LEFTMOST OPEN POSITION ON THE BOTTOM LEVEL OF THE TREE (OR TO START A NEW LEVEL IF THE BOTTOM LEVEL IS FULL). THIS ENTAILS INCREMENTING THE HEAP SIZE AND COPYING THE VALUE TO BE INSERTED TO THE NEW LAST ARRAY COMPONENT IN THE HEAP. (THIS WAS EXPLAINED IN SECTION 16.) IN THE IN-PLACE HEAP SORT, THE INCREMENTING IS HANDLED AUTOMATICALLY BY THE for LOOP AND THE VALUE TO BE INSERTED, Table (I), IS ALREADY AT THE RIGHT PLACE!

IN-PLACE HEAP SORT

- SOME OBSERVATIONS:

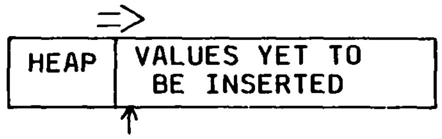
  - AFTER AN ITERATION OF THE FIRST LOOP, THERE ARE I VALUES ALREADY INSERTED IN THE HEAP AND Table'Length VALUES YET TO BE INSERTED. THE COMPONENTS OF Table WHOSE VALUES HAVE ALREADY BEEN INSERTED CAN BE CONSIDERED "EMPTY".

  - AFTER AN ITERATION OF THE SECOND LOOP, THERE ARE I VALUES COPIED BACK INTO Table AND Table'Length-I VALUES REMAINING IN Heap. THE COMPONENTS OF Table THAT HAVE NOT YET RECEIVED EXTRACTED VALUES CAN BE CONSIDERED "EMPTY".

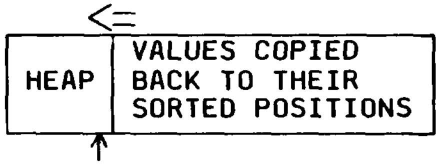
  - A HEAP CONTAINING n ELEMENTS CAN BE REPRESENTED BY n ARRAY COMPONENTS (AND A VARIABLE INDICATING THE CURRENT HEAP SIZE). THE HEAP GROWS AND SHRINKS FROM ITS RIGHT END.

- IF A LARGE ARRAY IS TO BE SORTED AND STORAGE SPACE IS SCARCE, THE "EMPTY" PART OF THE ARRAY CAN BE USED TO HOLD THE HEAP:

```
for I in 1 .. Table'Last loop
   -- Table (1 .. I-1) HOLDS THE HEAP
   -- Table (I .. Table'Last) HOLDS YET-TO-BE-INSERTED VALUES.
    INSERT Table (I) IN THE HEAP;
end loop;
```



```
for I in reverse 1 .. Table'Last loop
   -- Table (1 .. I) HOLDS THE HEAP
   -- Table (I + 1 .. Table'Last) HOLDS COPIED-BACK VALUES
    EXTRACT THE LARGEST VALUE FROM THE HEAP AND
    PLACE IT IN X                               ;
   -- NOW THE HEAP OCCUPIES Table (1 .. I-1).
   Table (I) := X;
end loop;
```



INSTRUCTOR NOTES

IN-PLACE HEAP SORT (CONTINUED)

- NOTES:
  - IN THIS APPROACH, VALUES TO APPEAR LATER IN THE SORTED ARRAY HAVE HIGHER PRIORITY.
  - THE HEAP OPERATIONS USE THE LOOP PARAMETER, I, TO DETERMINE THE CURRENT HEAP SIZE.
  - WE HAVE ASSUMED Table'First = 1. OTHERWISE, HEAP OPERATIONS CAN'T BE IMPLEMENTED AS DESCRIBED IN SECTION 16.

INSTRUCTOR NOTES

SECTION 18

LINKED LIST IMPLEMENTATION OF SETS

INSTRUCTOR NOTES

A SAMPLE BOOLEAN ARRAY IMPLEMENTATION WAS GIVEN EARLIER. A GENUINE PACKAGE WILL BE GIVEN AS AN EXERCISE.

THE LINKED LIST IMPLEMENTATION WILL PROVIDE US WITH A MORE GENERAL SET CAPABILITY BY ALLOWING NON-DISCRETE TYPES.

MERGEABLE SETS PROVIDE A WHOLE DIFFERENT WAY OF LOOKING AT SETS.

SETS

- STANDARD SETS
  - BOOLEAN ARRAY IMPLEMENTATION (SEEN IN SECTION 3)
  - LINKED LIST IMPLEMENTATION (THIS SECTION)
- MERGEABLE SETS
  - TREE IMPLEMENTATION (TO BE SEEN IN SECTION 19)

INSTRUCTOR NOTES

EMPHASIZE THAT THESE ARE THE TYPES OF SITUATIONS THAT THE PACKAGE DESIGNER MUST KEEP IN MIND.

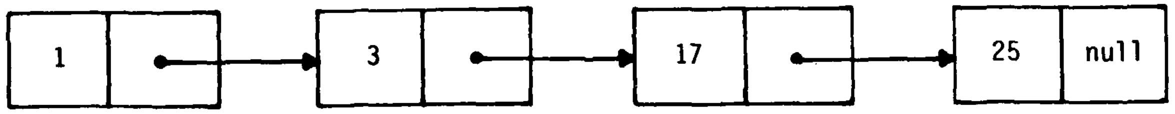
LINKED LIST VERSION

- GENERALIZES PREVIOUS PACKAGE BY PROVIDING FOR
  - SETS OF NON-DISCRETE ELEMENTS (REALS)
  - SETS OF NON-SCALAR ELEMENTS (TREES, SETS, ETC.)
- PAYING FOR GENERALIZATION WITH LESS EFFICIENT IMPLEMENTATION
- NON-COMPATIBLE WITH PACKAGE DEVELOPED AS A LAB EXERCISE:
  - CANNOT HAVE Complement OR Set_Range PROVIDED BY PACKAGE SINCE UNIVERSE OF DISCOURSE CANNOT NECESSARILY BE ENUMERATED.

INSTRUCTOR NOTES

ORDERED LISTS ARE USED TO MAKE PROCESSING MORE EFFICIENT, AS WE WILL SEE LATER.

REPRESENTING SETS BY LINKED LISTS



ORDERED LIST

{1, 3, 17, 25}

INSTRUCTOR NOTES

SOME SET OPERATIONS

- procedure Copy_Set (From : in Set_Type; To : out Set_Type);

  -- WANT TO COPY ELEMENTS IN SET, NOT POINTER TO THE SET

- function "=" (Left, Right : Set_Type) return Boolean;

  -- TWO SETS ARE EQUAL IF AND ONLY IF THEY HAVE THE SAME MEMBERS, NOT IF THE POINTERS TO THE SETS ARE THE SAME.

  -- THE Set_Type MUST BE LIMITED PRIVATE IF WE WANT TO PROVIDE "=". BUT THIS IS O.K., SINCE WE ARE PROVIDING AN ASSIGNMENT OPERATION (COPY).

INSTRUCTOR NOTES

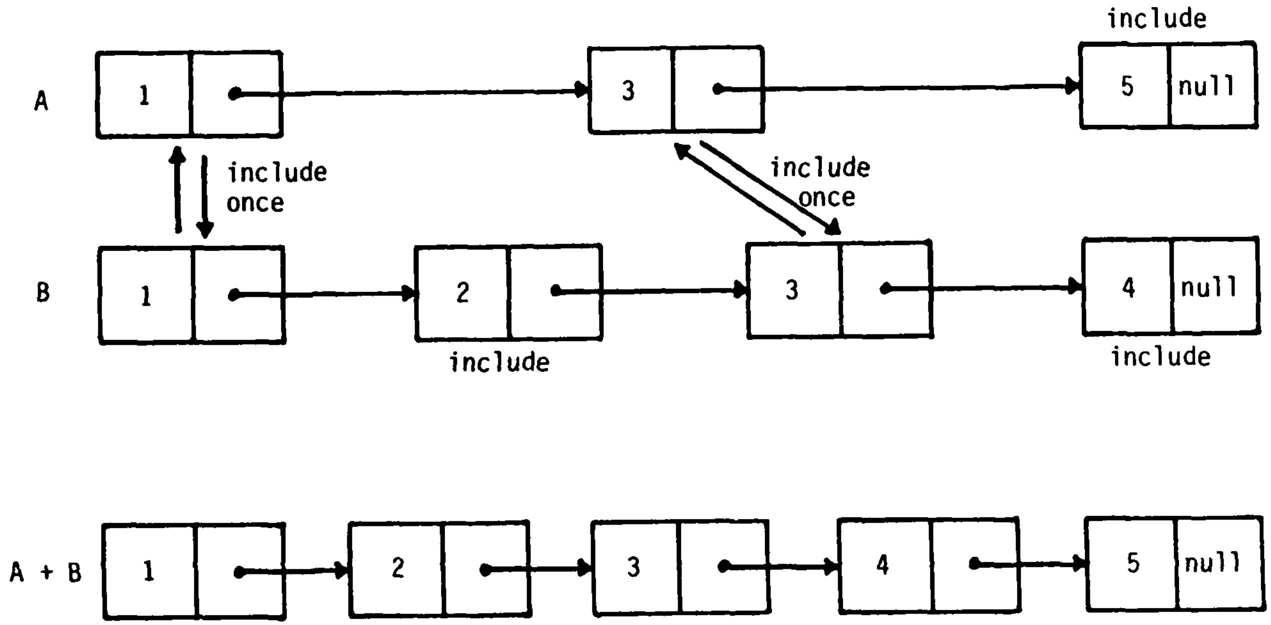
THE ENTRIES IN THE TABLE WILL REFLECT THE RESULT OF APPLYING THE OPERATION LISTED TO THE NEXT CELLS IN THE LISTS A AND B. THE FIRST COLUMN ASKS TO WHICH LIST A VALUE IS APPENDED. THE SECOND AND THIRD COLUMNS ARE YES/NO QUESTIONS.
ALGORITHM:

$\begin{pmatrix}\text{NEXT}\\\text{CELL}\\\text{OF}\\\text{A}\end{pmatrix} : \begin{pmatrix}\text{NEXT}\\\text{CELL}\\\text{OF}\\\text{B}\end{pmatrix}$	VALUE APPENDED TO RESULT	ADVANCE TO NEXT CELL OF A?	ADVANCE TO NEXT CELL OF B?
<	A's	YES	NO
=	EITHER	YES	YES
>	B's	NO	YES

WHEN ONE LIST IS EXHAUSTED, REMAINING ELEMENTS OF THE OTHER LIST ARE APPENDED TO THE RESULT.

IF THE LISTS WERE NOT ORDERED, WE WOULD HAVE TO SEARCH THROUGH THE SET WE WERE BUILDING EACH TIME WE TRIED TO ADD A NEW MEMBER TO THE SET (TO MAKE SURE IT WASN'T ALREADY THERE).

UNION



ALGORITHM:

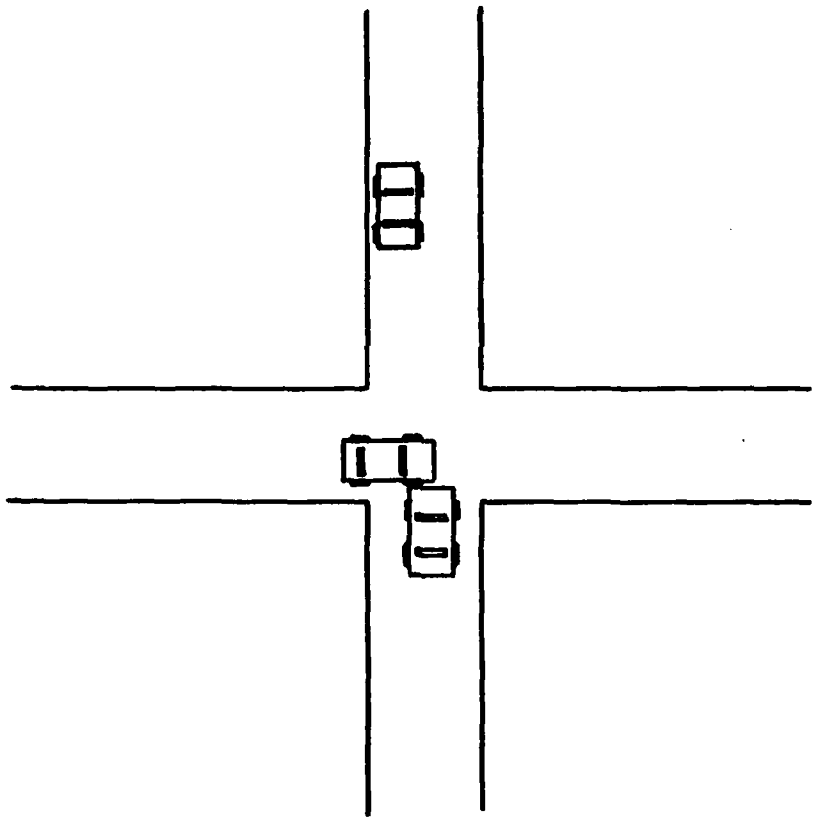
(NEXT CELL OF A) : (NEXT CELL OF B)	VALUE APPENDED TO RESULT	ADVANCE TO NEXT CELL OF A?	ADVANCE TO NEXT CELL OF B?
<			
=			
>			

WHAT HAPPENS WHEN THE END OF A LIST IS REACHED?

INSTRUCTOR NOTES

"SEE, COLLISIONS DON'T JUST HAPPEN IN HASHING."

INTERSECTION



INSTRUCTOR NOTES

BUT SERIOUSLY, FOLKS ...

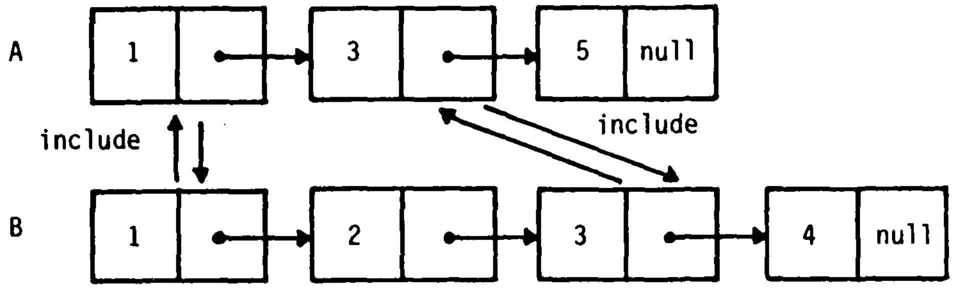
THIS TIME ONLY THE ELEMENTS THAT ARE IN BOTH LISTS ARE INCLUDED. THE ALGORITHM IS:

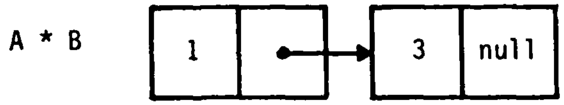
(NEXT CELL OF A) : (NEXT CELL OF B)	VALUE APPENDED TO RESULT	ADVANCE TO NEXT CELL OF A?	ADVANCE TO NEXT CELL OF B?
<	NONE	YES	NO
=	EITHER	YES	YES
>	NONE	NO	YES

THE COMPUTATION IS COMPLETE AS SOON AS ONE LIST IS EXHAUSTED.

NOTICE AGAIN HOW THE ORDERED LINKED LIST ALLOWS US TO PROCESS THE LISTS MORE REASONABLY.

INTERSECTION





ALGORITHM:

(NEXT CELL OF A) : (NEXT CELL OF B)	VALUE APPENDED TO RESULT	ADVANCE TO NEXT CELL OF A?	ADVANCE TO NEXT CELL OF B?
<			
=			
>			

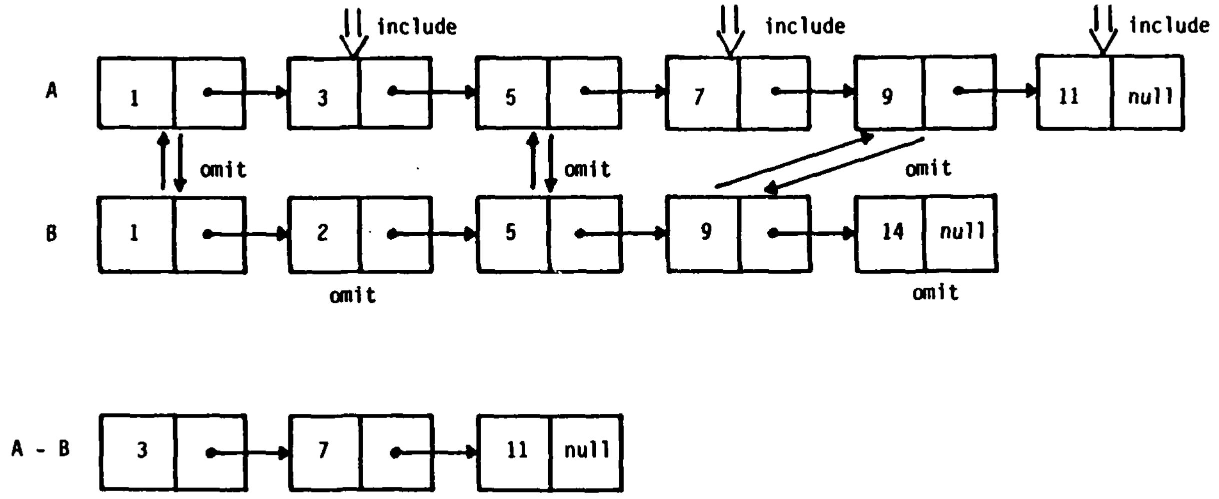
WHAT HAPPENS WHEN THE END OF A LIST IS REACHED?

INSTRUCTOR NOTES

(NEXT CELL OF A) : (NEXT CELL OF B)	VALUE APPENDED TO RESULT	ADVANCE TO NEXT CELL OF A?	ADVANCE TO NEXT CELL OF B?
<	A's	YES	NO
=	NONE	YES	YES
>	NONE	NO	YES

IF A IS EXHAUSTED FIRST, OR IF BOTH LISTS ARE EXHAUSTED AT THE SAME TIME, THE COMPUTATION IS COMPLETE. IF B IS EXHAUSTED FIRST, THE REMAINING ELEMENTS OF A ARE APPENDED TO THE RESULT.

SET DIFFERENCE



ALGORITHM:

(NEXT CELL OF A) : (NEXT CELL OF B)	VALUE APPENDED TO RESULT	ADVANCE TO NEXT CELL OF A?	ADVANCE TO NEXT CELL OF B?
<			
=			
>			

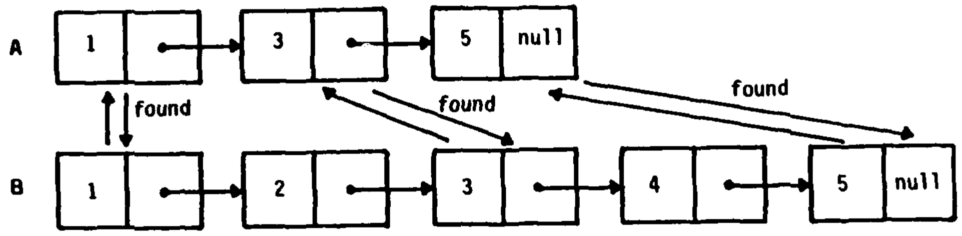
WHAT HAPPENS WHEN

- THE END OF A IS REACHED FIRST?
- THE END OF B IS REACHED FIRST?
- BOTH ENDS ARE REACHED AT THE SAME TIME?

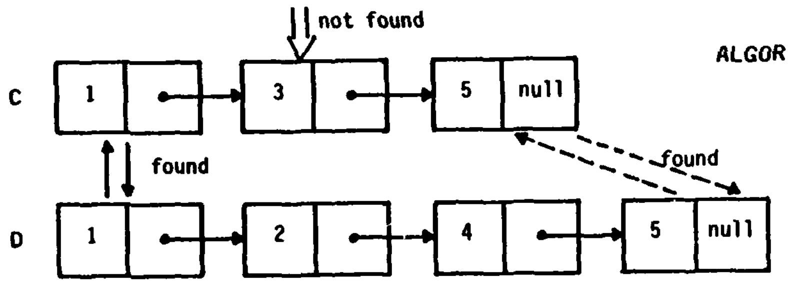
(NEXT CELL OF A) : (NEXT CELL OF B)	RETURN TRUE, RETURN FALSE, OR CONTINUE	ADVANCE TO NEXT CELL OF A?	ADVANCE TO NEXT CELL OF B?
<	RETURN False	-	-
=	EITHER	YES	YES
>	CONTINUE	YES	NO

IF B IS EXHAUSTED FIRST, RETURN False. OTHERWISE RETURN True.

SUBSET



A <= B : True



C <= D : False

ALGORITHM:

(NEXT CELL OF A) : (NEXT CELL OF B)	RETURN TRUE, RETURN FALSE, OR CONTINUE	ADVANCE TO NEXT CELL OF A?	ADVANCE TO NEXT CELL OF B?
<			
=			
>			

WHAT HAPPENS WHEN

- THE END OF A IS REACHED FIRST?
- THE END OF B IS REACHED FIRST?
- BOTH ENDS ARE REACHED AT THE SAME TIME?

INSTRUCTOR NOTES

EXAMPLES

● INTEGER SETS

● FIXED-POINT SETS

● FLOATING-POINT SETS

● SETS OF STRINGS

INSTRUCTOR NOTES

SECTION 19

MERGEABLE SETS

INSTRUCTOR NOTES

THE OPERATIONS ON MERGEABLE SETS ARE DIFFERENT FROM THOSE ON STANDARD SETS. THIS OCCURS BECAUSE THE INFORMATION WE NEED ABOUT MERGEABLE SETS IS DIFFERENT. WE DON'T NEED TO FORM THE INTERSECTION OR DIFFERENCE OF TWO MERGEABLE SETS.

FIRST Merge_Sets EXAMPLE:

ORIGINALLY, WE MIGHT HAVE KNOWN THAT FILES C, D, AND E HAD TO RESIDE ON THE SAME DISK AS FILE A, AND FILES F AND G HAD TO RESIDE ON THE SAME DISK AS FILE B. UPON DETERMINING THAT A AND B MUST RESIDE ON THE SAME DISK, WE MERGE THE SET {A, C, D, E} WITH THE SET {B, F, G} OBTAINING {A, B, C, D, E, F, G}. IT NOW FOLLOWS, FOR EXAMPLE, THAT FILES E AND F MUST RESIDE ON THE SAME DISK.

MERGEABLE SETS

- A DIFFERENT VIEW OF SETS
- TWO OBJECTS ARE IN THE SAME SET IF AND ONLY IF A CERTAIN RELATIONSHIP HOLDS BETWEEN THEM.
- OPERATIONS
  - Same_Set
    - -- MUST FILE A AND FILE B RESIDE ON THE SAME DISKETTE?
    - -- ARE NEW YORK AND WASHINGTON CONNECTED BY ACME AIRLINES FLIGHTS?
  - Merge_Sets
    - -- FILE A AND FILE B MUST RESIDE ON THE SAME DISKETTE, SO MERGE THE SET OF FILES THAT MUST RESIDE ON THE SAME DISKETTE AS A WITH THE SET OF FILES THAT MUST RESIDE ON THE SAME DISKETTE AS B.
    - -- A NEW FLIGHT CONNECTS NEW YORK AND WASHINGTON, SO MERGE THE SET OF CITIES THAT WERE CONNECTED WITH WASHINGTON AND THE SET OF CITIES THAT WERE CONNECTED WITH NEW YORK.

INSTRUCTOR NOTES

TWO CITIES ARE CONNECTED IF AND ONLY IF THEY ARE IN THE SAME SET.

THE QUESTION/ANSWER ILLUSTRATES Same_Set.

THE EXPANSION ILLUSTRATES Merge_Sets.

AN AIRLINE ROUTE EXAMPLE

TWO CITIES ARE IN THE SAME SET ONLY IF THEY ARE CONNECTED (DIRECTLY OR INDIRECTLY) BY ACME AIRLINES FLIGHTS.

ACME AIRLINES ONLY HAS FLIGHTS CONNECTING

-- NEW YORK WITH PHILADELPHIA
-- BALTIMORE WITH WASHINGTON

QUESTION: ARE NEW YORK AND WASHINGTON CONNECTED BY ACME AIRLINES FLIGHTS?
ANSWER: NO! THEY ARE IN DIFFERENT SETS.

{NEW YORK, PHILADELPHIA} {BALTIMORE, WASHINGTON}

ACME AIRLINES EXPANDS BY CONNECTING:

-- PHILADELPHIA WITH BALTIMORE (MERGING THEIR SETS)

QUESTION: ARE NEW YORK AND WASHINGTON CONNECTED BY ACME AIRLINES FLIGHTS?
ANSWER: YES! THEY ARE IN THE SAME SET.

{NEW YORK, PHILADELPHIA, BALTIMORE, WASHINGTON}

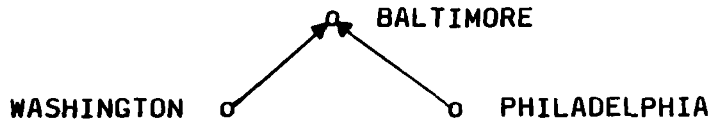
INSTRUCTOR NOTES

WE ARE IMPLEMENTING MERGEABLE SETS USING TREES.

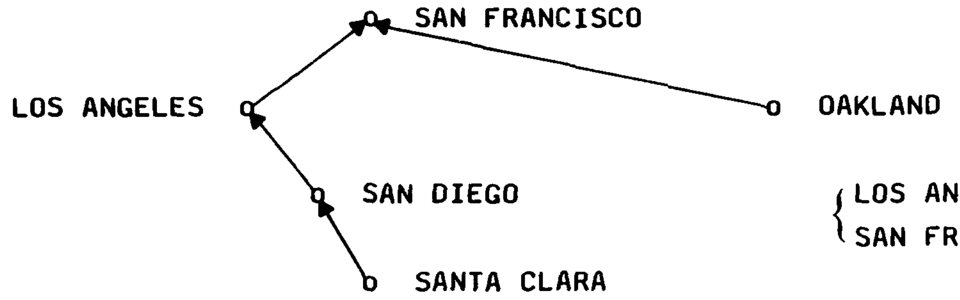
WE FINALLY GET AN EXAMPLE OF TREES WHERE ONLY POINTERS TO THE PARENTS ARE USED.

THE IMPLEMENTATION

- TREES
  - CHILDREN POINT TO PARENTS
  - PARENTS DO NOT POINT TO CHILDREN
- EXAMPLES



{BALTIMORE, PHILADEPHIA,
WASHINGTON}



{LOS ANGELES, OAKLAND, SAN DIEGO,
SAN FRANCISCO, SANTA CLARA}

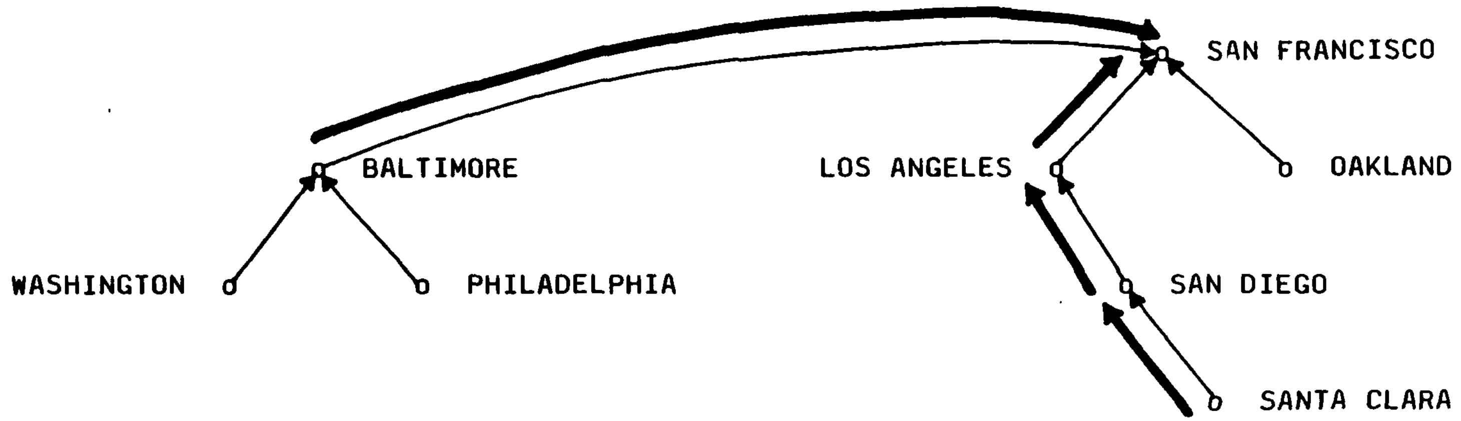
- MERGING MAKES ONE TREE A SUBTREE OF THE OTHER.
- TWO ELEMENTS ARE IN THE SAME SET IF AND ONLY IF THEY ARE IN THE SAME TREE.
- POSITION IN THE TREE IS IRRELEVANT.

INSTRUCTOR NOTES

TO SEE IF TWO CITIES ARE CONNECTED (IN THE SAME SET) WE START AT THOSE CITIES AND TRAVEL THE TREE UNTIL WE REACH THE ROOT. IF THEY HAVE THE SAME NODE, THEN THEY ARE IN THE SAME SET.

THIS IS THE REASON FOR THE POINTERS TO PARENTS.

Same_Set -- EXAMPLE



- ARE BALTIMORE AND SANTA CLARA CONNECTED?

- YES, SINCE THEY ARE IN THE SAME TREE.
  (FOLLOW POINTERS TO PARENTS UNTIL THE ROOT IS REACHED. TWO NODES ARE IN THE SAME TREE IF AND ONLY IF THEY HAVE THE SAME ROOT.)

INSTRUCTOR NOTES

WE MERGE THE SETS CONTAINING PHILADELPHIA AND SANTA CLARA.

WE CAN MERGE IN ONE OF TWO WAYS:

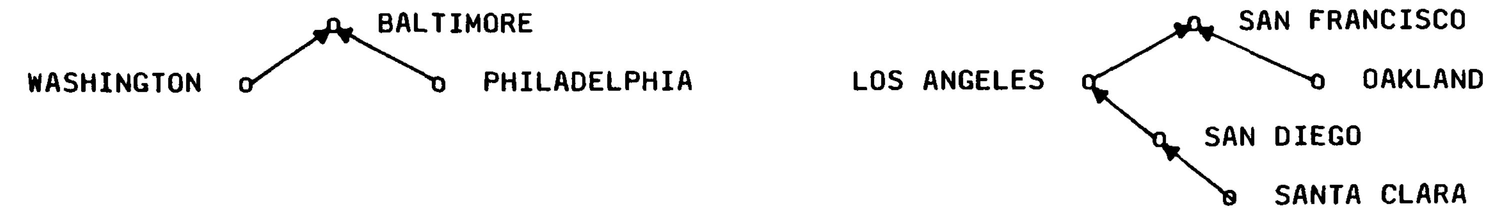
- MAKE THE TREE CONTAINING SANTA CLARA A SUBTREE OF THE TREE CONTAINING PHILADELPHIA, OR
- MAKE THE TREE CONTAINING PHILADELPHIA A SUBTREE OF THE TREE CONTAINING SANTA CLARA

THE SECOND FIGURE SHOWS THE RESULT OF THE FIRST CHOICE. WE JUST POINT THE ROOT OF SANTA CLARA'S TREE TO THE ROOT OF PHILADELPHIA'S TREE. THE THIRD FIGURE SHOWS THE OTHER CHOICE.

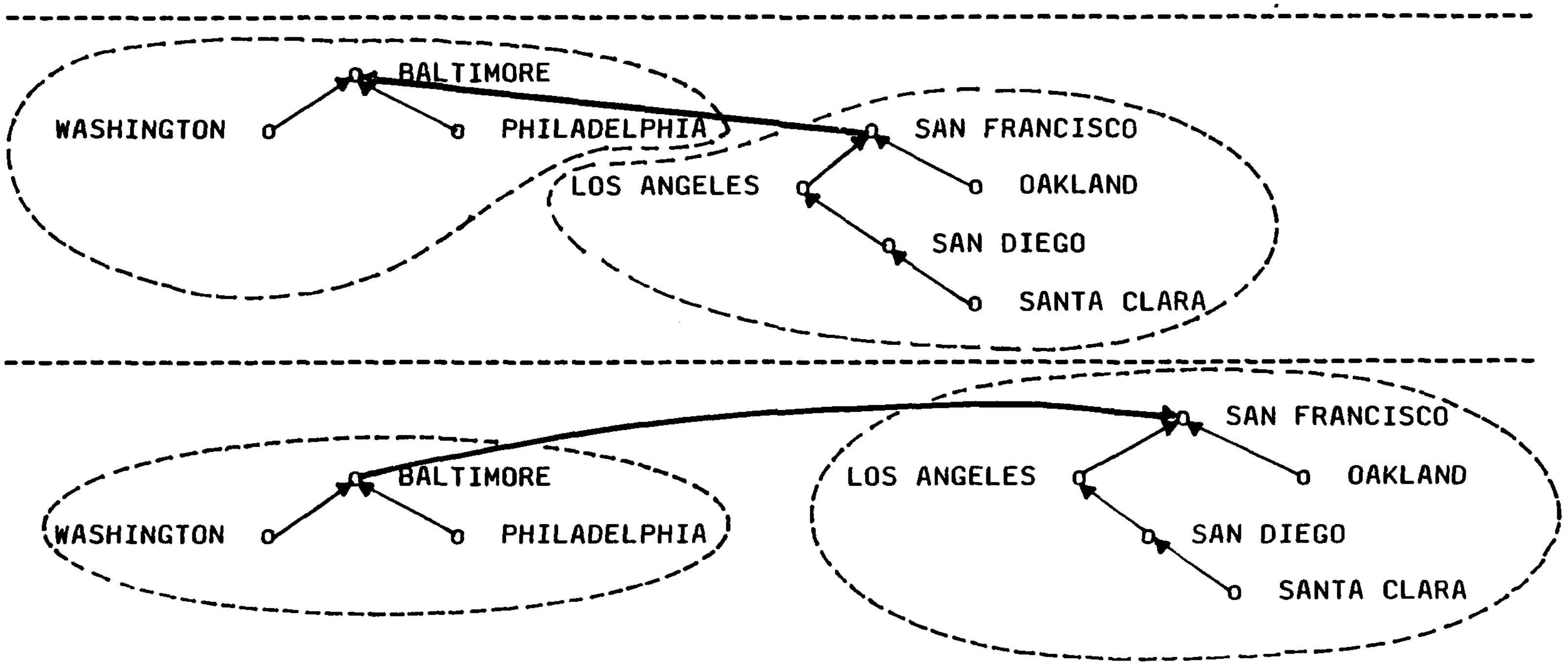
NOTICE THAT THE HEIGHT OF THE TREE IN THE THIRD FIGURE IS LESS THAN THE HEIGHT OF THE TREE IN THE SECOND FIGURE. SINCE Same_Set MUST TRAVEL A PATH IN THE TREE TO REACH THE ROOT, WE PREFER TO HAVE THE SHALLOWER TREE.

TO MINIMIZE THE AVERAGE DISTANCE FROM A NODE TO THE ROOT, WE KEEP TRACK OF THE NUMBER OF NODES IN EACH TREE. WHEN MERGING TWO SETS, THE TREE WITH THE SMALLER COUNT IS MADE A SUBTREE OF THE LARGER TREE.

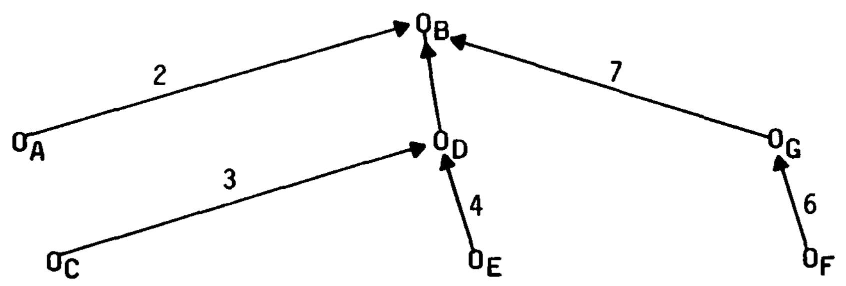
Merge_Sets -- EXAMPLE



- TWO WAYS TO MERGE PHILADELPHIA'S SET WITH SANTA CLARA'S SET:



INSTRUCTOR NOTES

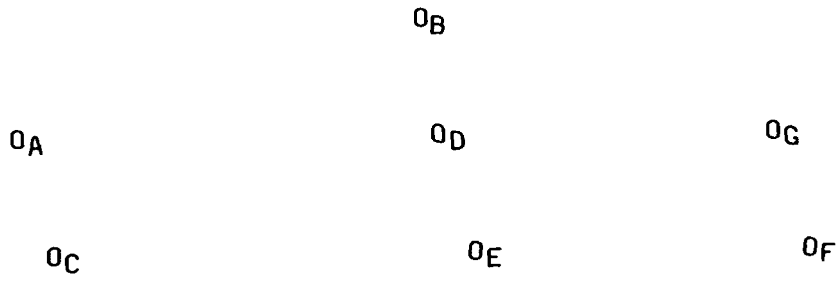


5. NO. A's ROOT IS B, E's ROOT IS D.

9. YES. A's ROOT AND E's ROOT ARE BOTH B.

EXERCISE

1. START WITH SEVEN ELEMENTS NAMED A, B, C, D, E, F, G:



2. MERGE A's SET WITH B's SET. (DRAW THE ARROW)
3. MERGE C's SET WITH D's SET. (DRAW THE ARROW)
4. MERGE E's SET WITH C's SET. (DRAW THE ARROW)
5. ARE A AND E IN THE SAME SET? (WHAT ARE THEIR ROOTS?)
6. MERGE F's SET WITH G's SET. (DRAW THE ARROW)
7. MERGE F's SET WITH A's SET. (DRAW THE ARROW)
8. MERGE D's SET WITH G's SET. (DRAW THE ARROW)
9. ARE A AND E THE SAME SET? (WHAT ARE THEIR ROOTS?)

INSTRUCTOR NOTES

THE Tree_Size_Part COMPONENT IS USED ONLY IN MERGING, AS WE WILL SEE. FOR ROOTS OF TREES, IT GIVES THE NUMBER OF NODES IN THE TREE. FOR OTHER NODES, ITS VALUE IS MEANINGLESS.

THE OPERATIONS WILL BE EXPLAINED AS WE GO ALONG.

THIS IS AN INVERTED VIEW OF SETS: Set_Type IS USED AS A COMPONENT OF A RECORD CONTAINING CITY DATA, THUS THE APPLICATION FOR CITIES IMPORT THE PACKAGE Mergeable_Sets_Package. WE ARE NOT INTERESTED IN THE MEMBERS OF A SET, BUT IN WHETHER TWO CITIES BELONG TO THE SAME SET.

WE ALWAYS START WITH A SINGLETON SET AND CONSTRUCT A SET BY MERGING IT WITH OTHER SETS.

Mergeable_Sets_Package SPECIFICATION

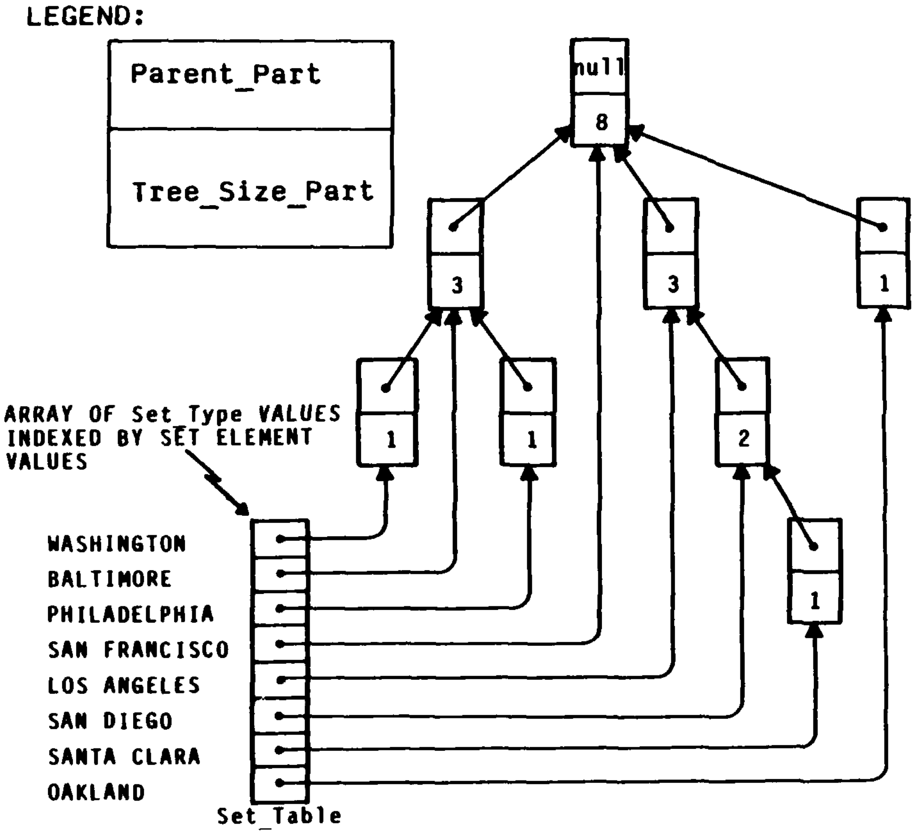
```
package Mergeable_Sets_Package is

   type Set_Type is private;

   function Same_Set (A, B : Set_Type) return Boolean;
   procedure Merge_Sets (A, B : in out Set_Type);

private

   type Set_Cell_Type;
   type Set_Pointer_Type is access Set_Cell_Type;

   type Set_Cell_Type is
      record
         Parent_Part : Set_Pointer_Type;
         Tree_Size_Part : Natural;
      end record;

   type Set_Type is
      record
         Set_Pointer_Part :
            Set_Pointer_Type :=
               new Set_Cell_Type (Parent_Part => null, Tree_Size_Part => 1);
               -- REPRESENTATION OF A ONE-ELEMENT SET
   -- A ONE-COMPONENT RECORD ALLOWS SPECIFICATION OF A DEFAULT INITIAL VALUE
      end record;

end Mergeable_Sets_Package;
```

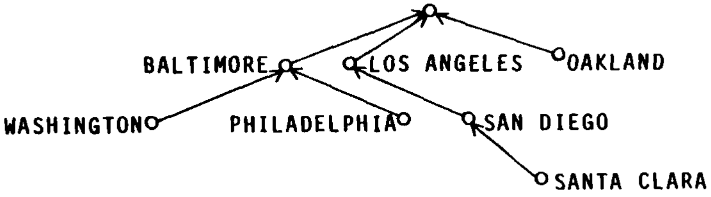
INSTRUCTOR NOTES

THE STRUCTURE ON THE LEFT IS THE ACTUAL REPRESENTATION OF THE TREE ON THE RIGHT. TO LOCATE THE NODE CORRESPONDING TO BALTIMORE, FOR EXAMPLE, WE USE THE Set_Type VALUE IN Set_Table (Baltimore). TWO CITIES C1 AND C2 ARE IN THE SAME SET IF AND ONLY IF NODES Set_Table (C1) AND Set_Table (C2) ARE NODES IN THE SAME TREE.

GIVEN AN ELEMENT, Set_Table ALLOWS US TO FIND THE ROOT OF THE SET CONTAINING THAT ELEMENT. THIS IS ALL WE REALLY HAVE TO DO FOR EITHER Same_Set OR Merge_Sets.

THE DATA STRUCTURE





INSTRUCTOR NOTES

THE PROCEDURES Merge_Sets AND Same_Set WORK EXACTLY AS DESCRIBED IN THE EXAMPLES.

THEY USE A SEPARATELY COMPILED FUNCTION Root_Element TO ACTUALLY TRAVEL THE TREE.

THERE IS NO WAY TO IMPROVE WHAT WE'VE DONE SO FAR, AS THE NEXT SLIDE WILL SHOW.

Mergeable_Sets_Package BODY

```
package body Mergeable_Sets_Package is

   function Root_Element (Set : Set_Pointer_Type) return Set_Pointer_Type
      is separate;

   function Same_Set (Set_1, Set_2 : in Set_Type) return Boolean is
      Root_1 : Set_Pointer_Type := Root_Element (Set_1.Set_Pointer_Part);
      Root_2 : Set_Pointer_Type := Root_Element (Set_2.Set_Pointer_Part);
   begin -- Same_Set
      return Root_1 = Root_2;
   end Same_Set;

   procedure Merge_Sets (Set_1, Set_2 : in out Set_Type) is

      Root_1 : Set_Pointer_Type := Root_Element (Set_1.Set_Pointer_Part);
      Root_2 : Set_Pointer_Type := Root_Element (Set_2.Set_Pointer_Part);

   begin -- Merge_Sets

   -- TO KEEP THE TREE SHALLOW, MAKE THE SMALLER TREE A SUBTREE OF THE LARGER

      if Root_1.Tree_Size_Part < Root_2.Tree_Size_Part then
      -- MAKE Root_1 A CHILD OF Root_2
         Root_1.Parent_Part := Root_2;
         Root_2.Tree_Size_Part := Root_1.Tree_Size_Part + Root_2.Tree_Size_Part;
      else
      -- MAKE Root_2 A CHILD OF Root_1
         Root_2.Parent_Part := Root_1;
         Root_1.Tree_Size_Part := Root_1.Tree_Size_Part + Root_2.Tree_Size_Part;
      end if;

   end Merge_Sets;

end Mergeable_Sets_Package;
```

INSTRUCTOR NOTES

WHEN WE FOLLOWED THE PATH FROM SANTA CLARA, WE ULTIMATELY REACHED THE ROOT. THIS IS ALL WE EVER WANT TO KNOW ABOUT A NODE. IF WE DO NOT ADJUST THE TREE WE LOSE THIS INFORMATION UNTIL THE NEXT TIME WE ASK ABOUT THE NODE. MOREOVER, SINCE WE ALSO VISITED SAN DIEGO, WE ALSO KNOW WHAT THE ROOT OF ITS SET IS.

WHAT WE DO INSTEAD IS ADJUST ANY NODE WE VISIT TO POINT TO THE ROOT OF ITS SETS.

THE NEXT FIGURE SHOWS THE RESULT.

THE Tree_Size_Part COMPONENTS FOR LOS ANGELES AND SAN DIEGO ARE NO LONGER CORRECT, BUT ONLY THE Tree_Size_Part OF THE ROOT IS IMPORTANT. THE COMPONENT IS USED ONLY DURING Merge_Sets, TO DETERMINE WHICH ROOT SHOULD BECOME A CHILD OF THE OTHER.

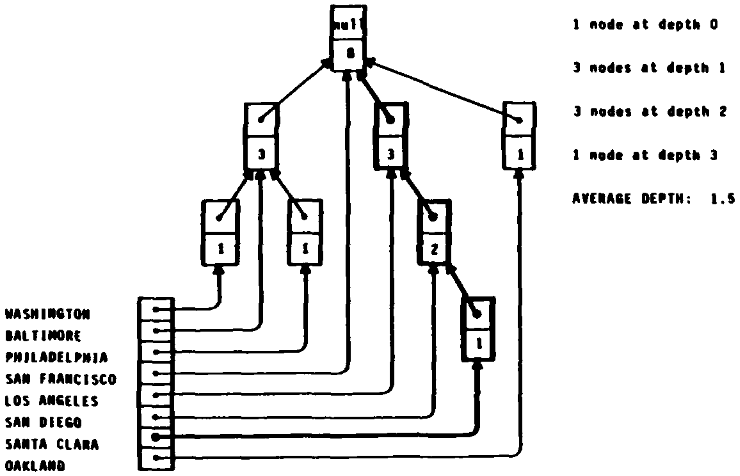
THE NEXT SLIDE SHOWS HOW THE READJUSTMENT IS ACHIEVED.

REDUCING TREE DEPTH WHEN LOCATING THE ROOT OF THE TREE

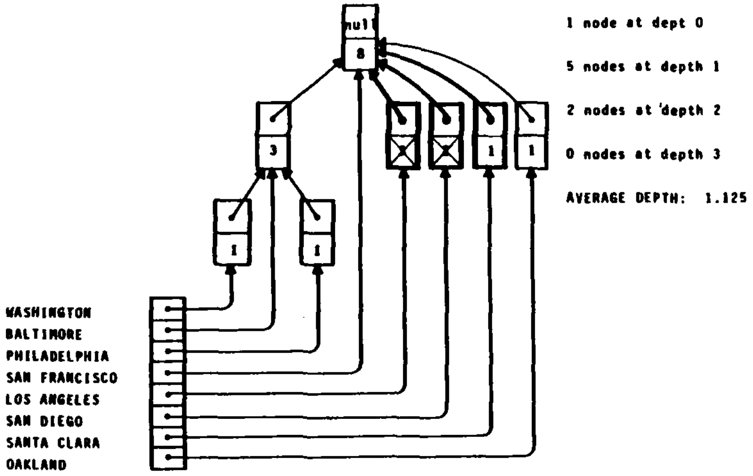
- Same_Set AND Merge_Set WILL WORK MORE QUICKLY WHEN THERE ARE FEWER LINKS TO FOLLOW FROM A NODE TO THE ROOT.
- STRATEGY FOR REDUCING DEPTH OF NODES:

  WHEN FOLLOWING THE PATH FROM A NODE TO THE ROOT, ADJUST EACH NODE ENCOUNTERED ALONG THE WAY TO POINT DIRECTLY TO THE ROOT.

BEFORE FINDING SANTA CLARA'S ROOT:



AFTER POINTING ALL NODES ON THAT PATH DIRECTLY TO THE ROOT:



- Tree_Size_Part VALUES REMAIN VALID FOR THE ROOT, BUT NOT NECESSARILY FOR OTHER NODES

INSTRUCTOR NOTES

NOTE THAT IT IS THE ASSIGNMENT AT THE RECURSIVE CALL THAT PROPAGATES THE ROOT BACK DOWN THE PATH.

THE SEQUENCE OF RECURSIVE CALLS FOLLOWS THE PATH FROM THE NODE UP TO THE ROOT. WHEN WE REACH THE ROOT WE TURN AROUND AND GO BACK DOWN THE PATH, MODIFYING Parent_Part COMPONENTS.

```
separate (Mergeable_Sets_Package)
function Root_Element (Set : Set_Pointer_Type) return Set_Pointer_Type is
begin -- Root_Element

   if Set.Parent_Part = null then   -- AT THE ROOT
      return Set;
   else
   -- CALL Root_Element RECURSIVELY WITH Set.Parent_Part, SINCE
   -- THE PARENT'S ROOT IS THIS NODE'S ROOT.
   -- ASSIGN THE RESULT OF THE CALL TO THIS NODE'S Parent_Part, SO THIS
   -- NODE POINTS DIRECTLY TO THE ROOT FROM NOW ON.
   -- (THE RECURSIVE CALL HAS THE SIDE-EFFECT OF ADJUSTING NODES HIGHER IN
   -- THE TREE TO POINT DIRECTLY TO THE ROOT.)
      Set.Parent_Part := Root_Element (Set.Parent_Part);
      return Set.Parent_Part;
   end if;

end Root_Element;
```

INSTRUCTOR NOTES

IN THE LITERATURE, THE Merge_Sets OPERATION IS CALLED Union AND THE Root_Element FUNCTION USED TO IMPLEMENT Merge_Sets AND Same_Set IS CALLED Find.

MERGEABLE SETS -- CONCLUSION

- AN EXAMPLE USING THE Mergeable_Set_Package WILL BE GIVEN IN SECTION 20.

- MERGEABLE SETS APPEAR IN THE LITERATURE AS "Union-Find" OR "Find-Union" TREES.

INSTRUCTOR NOTES

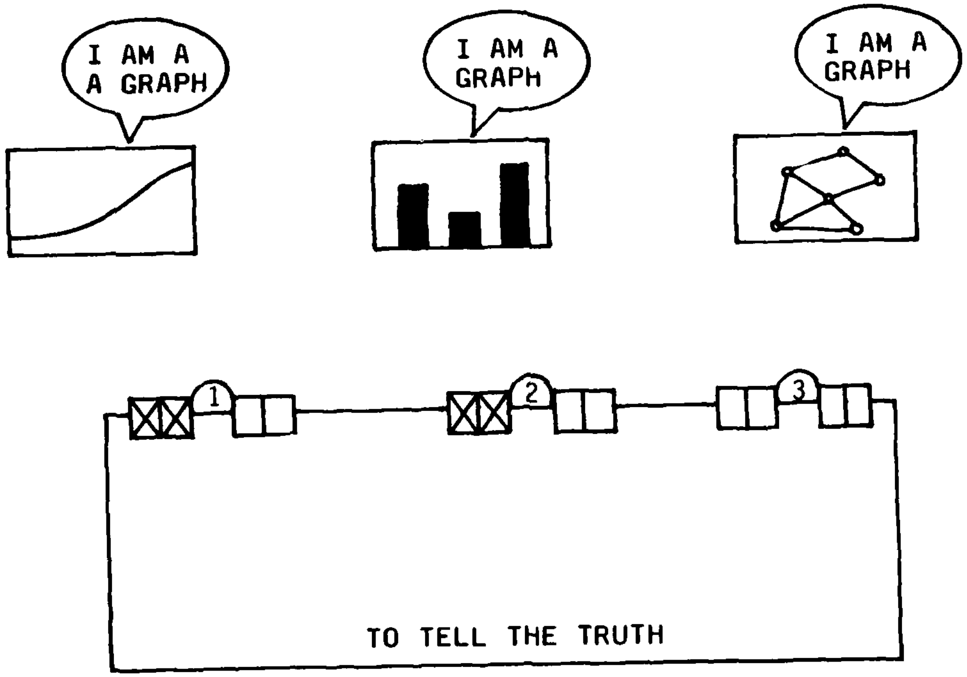
SECTION 20

GRAPHS

INSTRUCTOR NOTES

HINT: COMPUTER SCIENTISTS CONSIDER CONTESTANT #3 AS A GRAPH. (THEY ALSO DON'T EAT QUICHE!)

WILL THE REAL GRAPH PLEASE STAND UP?



INSTRUCTOR NOTES

THE PLURAL OF VERTEX IS VERTICES.

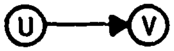
INTUITIVELY, A PATH FROM V_1 TO V_n IS A WAY TO GET FROM VERTEX V_1 TO V_n BY FOLLOWING EDGES IN THE GRAPH. IN A DIRECTED GRAPH, THE EDGES ARE ONE-WAY STREETS.

GRAPHS -- BASIC DEFINITIONS

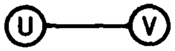
- A GRAPH CONSISTS OF
  - -- A SET OF VERTICES
  - -- A SET OF EDGES CONNECTING PAIR OF VERTICES

- TWO KINDS OF GRAPHS
  - -- DIRECTED: AN EDGE GOES FROM ONE VERTEX TO ANOTHER
  - -- UNDIRECTED: AN EDGE IS A SYMMETRIC CONNECTION BETWEEN TWO VERTICES

- IF THERE IS AN EDGE CONNECTING VERTICES U AND V, THEN WE SAY VERTICES U AND V ARE ADJACENT. WE DEPICT THIS AS

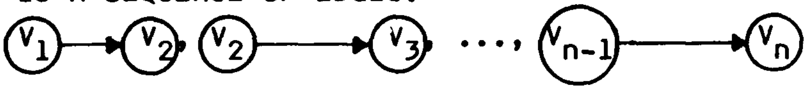


IF THE GRAPH IS DIRECTED; OTHERWISE WE DEPICT THIS AS



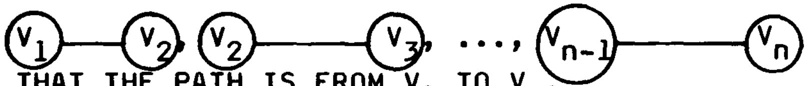
- IN A DIRECTED GRAPH, IF THERE IS AN EDGE FROM U TO V, V IS CALLED A SUCCESSOR OF U. (VERTEX U MAY HAVE MANY SUCCESSORS.)

- A PATH IS A SEQUENCE OF EDGES.



(FOR DIRECTED GRAPH)

OR

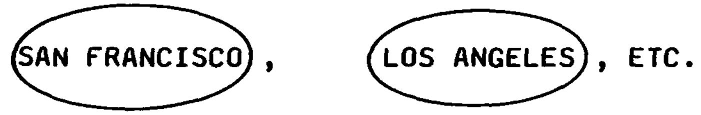


(FOR UNDIRECTED GRAPH)

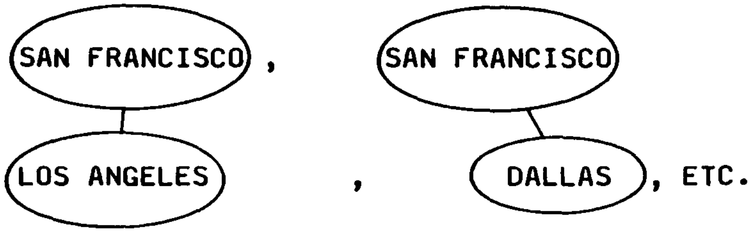
WE SAY THAT THE PATH IS FROM V_1 TO V_n.

THE NAME OF THE CITY ACTUALLY CORRESPONDS TO DATA. WE WILL TAKE A LOOK AT HOW DATA IS INCLUDED WHEN WE LOOK AT IMPLEMENTATIONS.

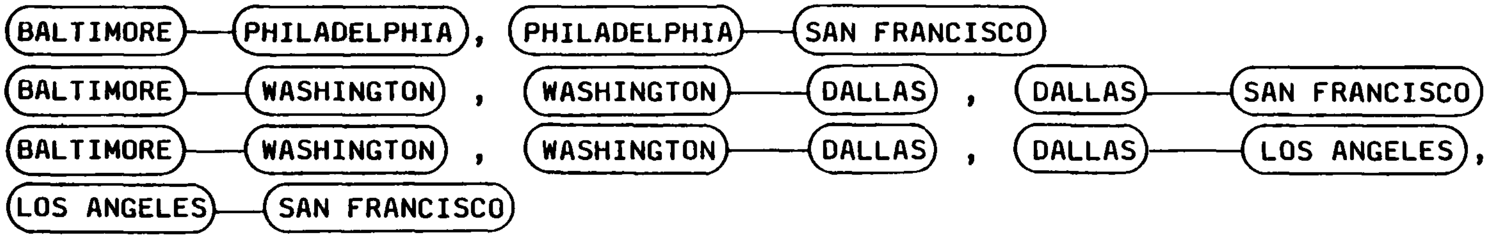
EXAMPLES OF VERTICES:



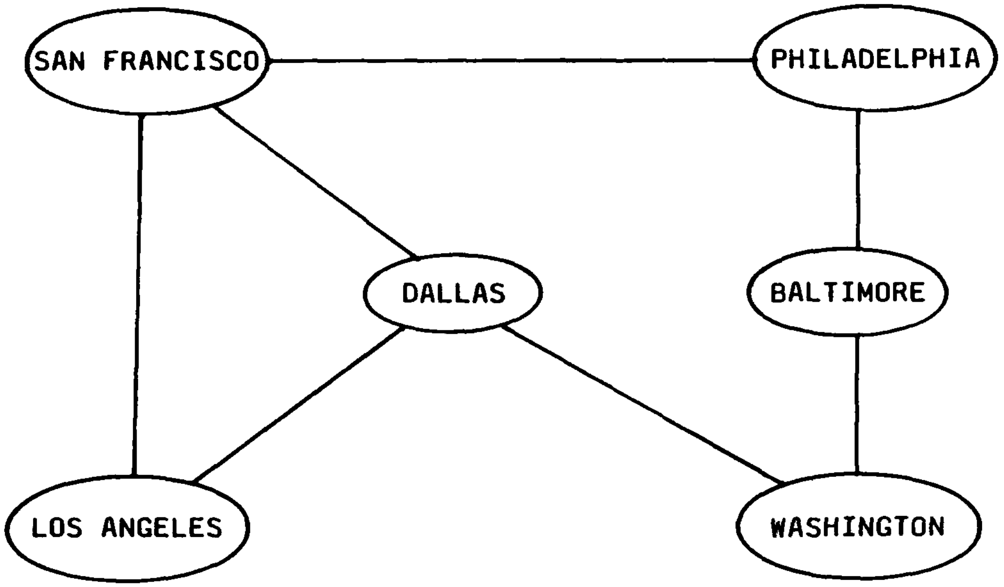
EXAMPLES OF EDGES:



EXAMPLES OF PATHS FROM BALTIMORE TO SAN FRANCISCO



UNDIRECTED GRAPH EXAMPLE



GIVE EXAMPLES OF:

1. A VERTEX

2. AN EDGE

3. TWO PATHS FROM BALTIMORE TO SAN FRANCISCO

- ACME AIRLINES FLIGHT CONNECTIONS
- TWO CITIES ARE DIRECTLY CONNECTED BY AN ACME AIRLINES FLIGHT IF AND ONLY IF THEIR VERTICES ARE ADJACENT.

  EXAMPLES:

  - ACME AIRLINES HAS DIRECT FLIGHTS BETWEEN BALTIMORE AND WASHINGTON AND BALTIMORE AND PHILADELPHIA.
  - ACME AIRLINES HAS A ROUTE (A PATH) FROM PHILADELPHIA TO WASHINGTON WITH A STOPOVER IN BALTIMORE.

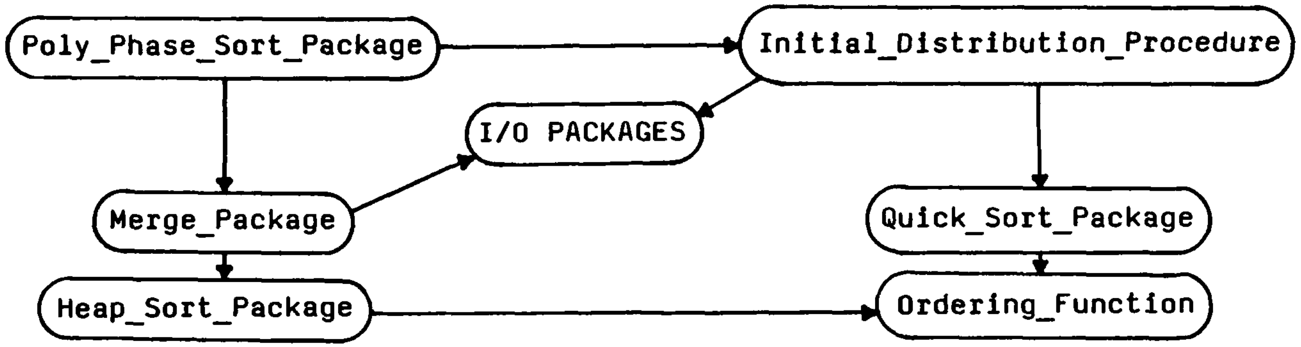
INSTRUCTOR NOTES

FYI - A Poly_Phase SORT IS AN EXTERNAL SORT. IT READS A "LARGE" CHUNK FROM AN INPUT FILE, USES QUICKSORT TO SORT THE CHUNK, AND WRITES IT TO AN OUTPUT FILE. IT CONTINUES TO DO SO UNTIL ALL INPUT HAS BEEN READ. THEN IT USES A KEYSORT TO MERGE THE FILES INTO ONE.

A BINARY RELATION OVER TWO SETS ASSOCIATES VALUES FROM THE SETS. EXAMPLE: GIVEN THE SET OF MONTHS AND SEASONS, WE COULD DEFINE A BINARY RELATION Is_A_Month_In_The_Season.

DIRECTED GRAPHS CORRESPOND TO BINARY RELATIONS WITH VERTICES CORRESPONDING TO SET VALUES.

DIRECTED GRAPH EXAMPLE



- COMPILATION DEPENDENCIES FOR COMPONENTS OF AN ADA PROGRAM
- DIRECTED EDGE FROM U TO V INDICATES THAT U IS DEPENDENT ON (MUST BE COMPILED LATER THAN) V.
- EXAMPLES:
  - -- Heap_Sort_Package AND Quick_Sort_Package BOTH REQUIRE THAT Ordering_Function BE COMPILED FIRST
  - -- Merge_Package REQUIRES THAT THE I/O PACKAGES AND Heap_Sort_Package (THEREFORE ALSO Ordering_Function) BE COMPILED FIRST.
- THIS FORMS A <u>BINARY RELATION</u> THAT COULD BE NAMED Compilation_Depends_On.

INSTRUCTOR NOTES

WEIGHTED GRAPHS CAN BE USED TO EXPRESS THE COST OF TRAVELING ALONG A PATH THROUGH A GRAPH.

WEIGHTED GRAPH

- A GRAPH IS A WEIGHTED GRAPH IF EACH EDGE HAS A "COST" (OR "WEIGHT") ASSOCIATED WITH IT.

- EXAMPLES OF COSTS ARE

  -- MILEAGE

  -- MANUFACTURING EXPENSES

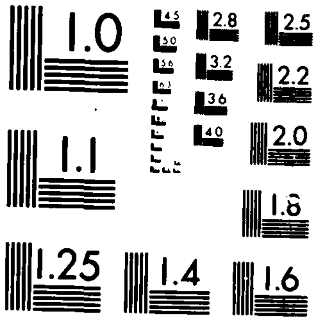
  -- RISK FACTORS

  -- COMPILATION TIME

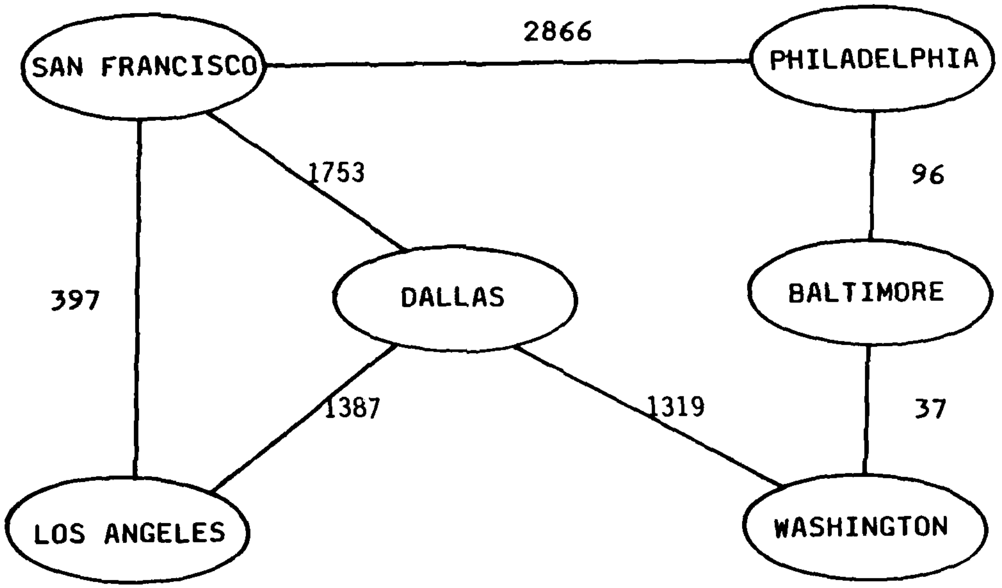
INSTRUCTOR NOTES

AD-A165 076 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 2(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506 7/7

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

A WEIGHTED GRAPH EXAMPLE



- ACME AIRLINES FLIGHT CONNECTIONS
- THIS GRAPH SHOWS THE MILEAGE BETWEEN CITIES
- EXAMPLE
  - -- YOU CAN GET TO LOS ANGELES FROM BALTIMORE WITH MINIMUM STOPOVERS BY STOPPING IN PHILADELPHIA AND SAN FRANCISCO OR BY STOPPING IN WASHINGTON AND DALLAS.
  - -- THE SHORTEST ROUTE IS FROM BALTIMORE TO WASHINGTON, DALLAS, AND THEN LOS ANGELES.

INSTRUCTOR NOTES

UPCOMING SLIDES DESCRIBE THESE THREE IMPLEMENTATIONS.

SOME IMPLEMENTATIONS OF GRAPHS

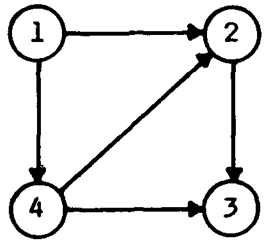
- ADJACENCY MATRICES
- SUCCESSOR LISTS
- EDGE SETS

INSTRUCTOR NOTES

WE ARE PRESENTING THE UNWEIGHTED VERSION FIRST FOR EASIER UNDERSTANDING.

AN EXAMPLE IS REACHABILITY, I.E., CAN I GET FROM ONE VALUE TO ANOTHER?

IMPLEMENTATION 1 -- ADJACENCY MATRIX



A:

	1	2	3	4
1	False	True	False	True
2	False	False	True	False
3	False	False	False	False
4	False	True	True	False

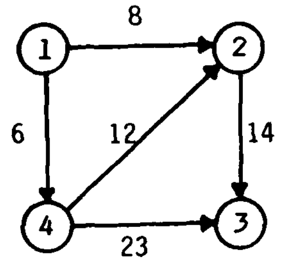
$$A\ (i,\ j) = \begin{cases} \text{True, IF THERE IS AN EDGE FROM } V_i \text{ TO } V_j \\ \text{False, OTHERWISE} \end{cases}$$

- FOR AN UNDIRECTED GRAPH, A (i, j) = A (j, i)

INSTRUCTOR NOTES

AN "EMPTY" MATRIX ENTRY IS NOT UNINITIALIZED. RATHER, IT CONTAINS A SPECIAL VALUE DISTINCT FROM ALL "FULL" MATRIX ENTRIES.

ADJACENCY MATRIX -- WEIGHTED VERSION



	1	2	3	4
1	-	8	-	6
2	-	-	14	-
3	-	-	-	-
4	-	12	23	-

- IF TWO VERTICES ARE ADJACENT THEN THE MATRIX ENTRY CONTAINS COST INFORMATION. OTHERWISE THE MATRIX ENTRY IS EMPTY.

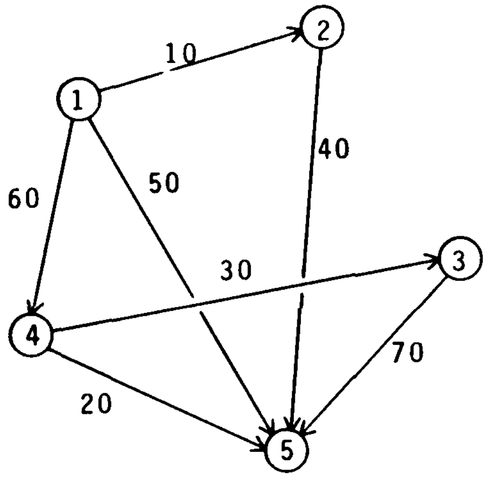
- AN ADJACENCY MATRIX IS GOOD FOR DIRECTED AND UNDIRECTED GRAPHS.

INSTRUCTOR NOTES

	1	2	3	4	5
1	-	10	-	60	50
2	-	-	-	-	40
3	-	-	-	-	70
4	-	-	30	-	20
5	-	-	-	-	-

EXERCISE

CONSTRUCT AN ADJACENT MATRIX FOR THE FOLLOWING WEIGHTED DIRECTED GRAPH:



INSTRUCTOR NOTES

NO COMMENT.

GRAPH IMPLEMENTATION

```
package Graph_Package is
   type Graph_Type is private;
private
```

WARNING:

THIS DEPICTION OF A PRIVATE PART
MAY BE TOO GRAPH-IC FOR SOME STUDENTS.

```
end Graph_Package;
```

INSTRUCTOR NOTES

IF THE GRAPH IS NOT WEIGHTED THEN THE MATRIX COMPONENTS ARE JUST BOOLEAN VALUES.

THE NEXT SLIDE SHOWS HOW DATA MAY BE ASSOCIATED WITH VERTICES.

TYPE DECLARATIONS FOR AN ADJACENCY MATRIX

```
type Vertex_Type is [some integer or enumeration type definition];

type Cost_Type is [some numeric type definition];

type Edge_Description_Type (Present : Boolean) is
   record
      case Present is
         when True =>
            Cost_Part : Cost_Type;
         when False =>
            null;
      end case;
   end record;

type Adjacency_Matrix is
   array (Vertex_Type, Vertex_Type) of Edge_Description_Type;
```

INSTRUCTOR NOTES

THE ABSTRACT TYPE Graph_Type PROVIDES A NAMING OF THE INTERVAL NODES. FOR EXAMPLE, CITIES ON THE ACME AIRLINES EXAMPLE.

THIS IS THE ONLY REPRESENTATION WE DO THIS FOR.

ASSOCIATING DATA WITH VERTICES

```
type Vertex_Data_Type is ...;
type Vertex_List_Type is
   array (Vertex_Type) of Vertex_Data_Type;
```

- Vertex_List_Type PROVIDES A WAY OF ASSOCIATING DATA WITH EACH VERTEX

```
type Graph_Type is
   record
      Vertex_Part : Vertex_List_Type;
      Edge_Part : Adjacency_Matrix_Type;
   end record;
```

- IF NO VERTEX DATA IS USED, THEN Graph_Type IS JUST THE ADJACENCY MATRIX.

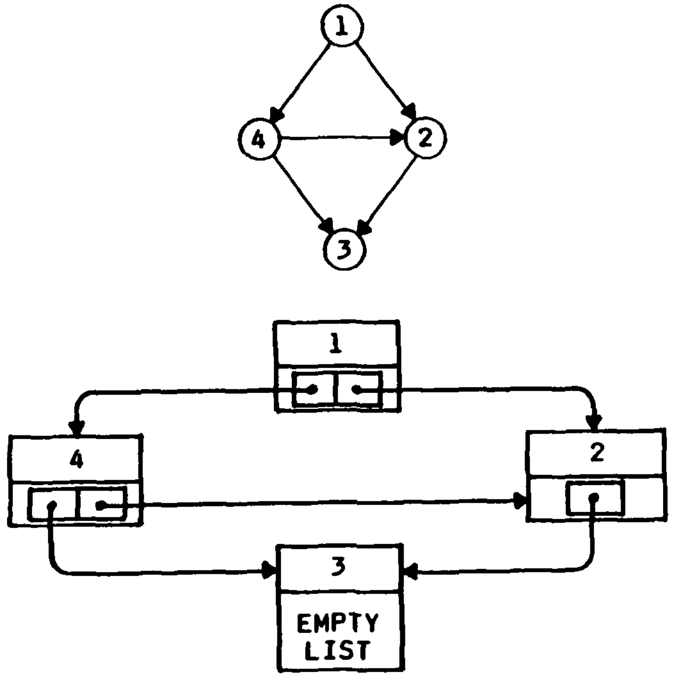
INSTRUCTOR NOTES

PRESENTING THE UNWEIGHTED VERSION FIRST FOR SIMPLICITY.

NOTICE THAT THE ADJACENCY LIST IS ACTUALLY PART OF THE VERTEX.

THE ARROWS REPRESENT DIRECTED EDGES (AN ABSTRACT MATHEMATICAL NOTION) IN THE UPPER DIAGRAM AND ACCESS VALUES (AN ADA NOTION) IN THE LOWER DIAGRAM.

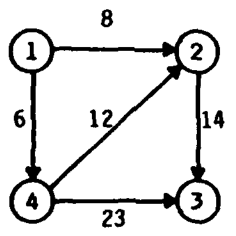
IMPLEMENTATION 2 -- SUCCESSOR LISTS



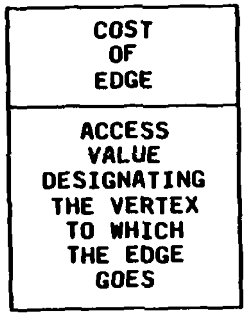
- REPRESENT VERTICES USING RECORDS
- EACH RECORD CONTAINS:
  - -- DATA ASSOCIATED WITH THE VERTEX
  - -- A LIST OF ACCESS VALUES POINTING TO THE SUCCESSORS OF THIS VERTEX. THE LISTS CAN BE IMPLEMENTED AS
    - A LINEAR LIST (AS SHOWN ABOVE), OR
    - A LINKED LIST
  - THIS REPRESENTATION IS BETTER FOR DIRECTED GRAPHS THAN FOR UNDIRECTED GRAPHS, SINCE YOU CAN ONLY FOLLOW THE REPRESENTATION OF AN EDGE IN ONE DIRECTION

INSTRUCTOR NOTES

SUCCESSOR LISTS -- WEIGHTED VERSION

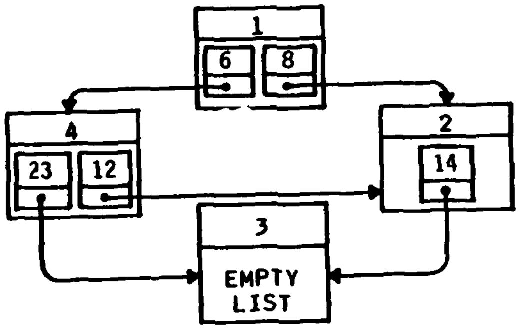


SUCCESSOR LIST ELEMENT:



COST
OF
EDGE

ACCESS
VALUE
DESIGNATING
THE VERTEX
TO WHICH
THE EDGE
GOES



VG 679.2

INSTRUCTOR NOTES

WE USE THE List_Package_Template YOU IMPLEMENTED AS AN EXERCISE.

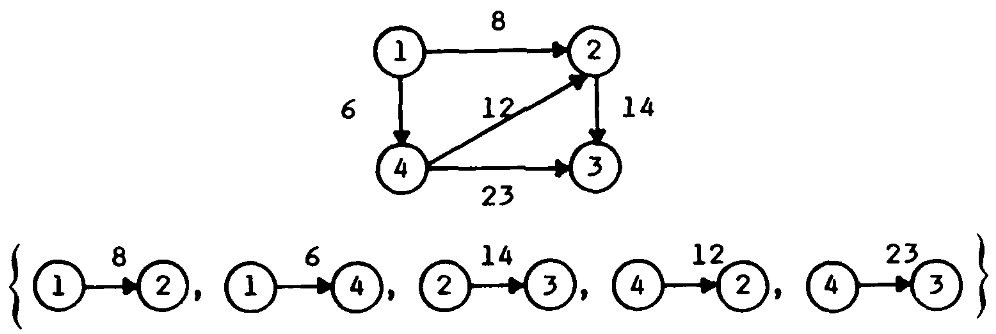
DATA TYPES FOR Successor_List_Package

```
type Vertex_Type;

type Vertex_Pointer_Type is access Vertex_Type;

type Edge_Type is
   record
      Cost_Part      : Cost_Type;
      Successor_Part : Vertex_Pointer_Type;
   end record;

package Successor_List_Package is
   new List_Package_Template (Edge_Type);

type Vertex_Type is
   record
      Data_Part           : Vertex_Data_Type;
      Successor_List_Part : Successor_List_Package.List_Type;
   end record;
```

INSTRUCTOR NOTES

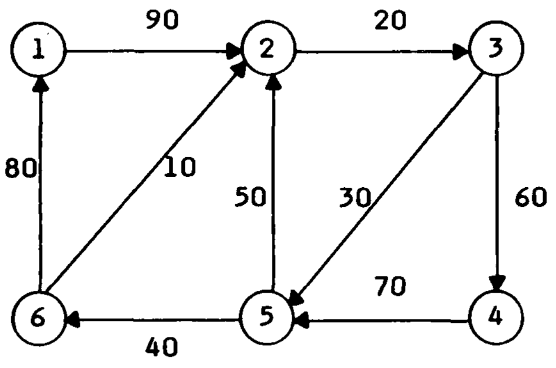
WE WILL SEE AN EXAMPLE OF ITS USE IN A FEW MINUTES.

IMPLEMENTATION 3 -- EDGE SET



- THIS IS SIMPLY A SET OF EDGES

- IT CAN BE USED WITH DIRECTED OR UNDIRECTED GRAPHS


1
2
3
4
5
6
90
20
80
10
50
30
60
70
40

EXERCISE

DRAW THE GRAPH REPRESENTED BY THE FOLLOWING EDGE SET:

$$\left\{ \textcircled{1} \xrightarrow{90} \textcircled{2},\ \textcircled{2} \xrightarrow{20} \textcircled{3},\ \textcircled{3} \xrightarrow{60} \textcircled{5},\ \textcircled{4} \xrightarrow{70} \textcircled{5},\ \textcircled{5} \xrightarrow{40} \textcircled{6},\ \textcircled{6} \xrightarrow{80} \textcircled{1},\right.$$

$$\left. \textcircled{6} \xrightarrow{10} \textcircled{2},\ \textcircled{5} \xrightarrow{50} \textcircled{2},\ \textcircled{3} \xrightarrow{30} \textcircled{5} \right\}$$

INSTRUCTOR NOTES

WE ARE USING A LINKED LIST GENERIC SET PACKAGE.

AN ORDERING RELATION IS NEEDED TO IMPLEMENT SETS AS LINKED LISTS, AS DESCRIBED IN SECTION 18.

ANY FUNCTION WILL DO AS LONG AS IT OBEYS THE FOLLOWING RULES:

1. FOR ANY EDGE e, Should_Be_Listed_Before (e, e) IS FALSE.

2. FOR ANY EDGES el AND e2,
   Should_Be_Listed_Before (el, e2) = not Should_Be_Listed_Before (e2, el)

3. FOR ANY EDGES el, e2, AND e3,
   Should_Be_Listed_Before (el, e2) and Should_Be_Listed_Before (e2, e3) implies
   Should_Be_Listed_Before (el, e3)

AN EXAMPLE IS

```
function Should_Be_Listed_Before (Edge_1, Edge_2 : Edge_Type) return Boolean is
begin
   return Edge_1.Vertex_1_Part < Edge_2.Vertex_1_Part or
         (Edge_1.Vertex_1_Part = Edge_2.Vertex_1_Part and Edge_1.Vertex_2_Part <
          Edge_2.Vertex_2_Part);
end Should_Be_Listed_Before;
```

TYPE DECLARATIONS FOR EDGE SETS

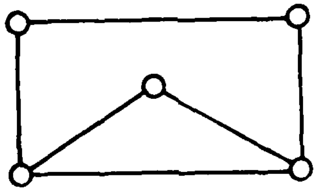
```
type Vertex_Type is [some integer or enumeration type definition];

type Cost_Type is [some numeric type definition];

type Edge_Type is
   record
      Vertex_1_Part, Vertex_2_Part : Vertex_Type;
      Cost_Part                    : Cost_Type;
   end record;

function Should_Be_Listed_First (Edge_1, Edge_2 : Edge_Type) return Boolean;
   -- CRITERION FOR BUILDING ORDERED LISTS TO REPRESENT SETS OF EDGES.

package Edge_Set_Package is
   new Linked_Set_Package_Template (Edge_Type, Should_Be_Listed_First);

subtype Edge_Set_Type is Edge_Set_Package.Set_Type;
```

INSTRUCTOR NOTES

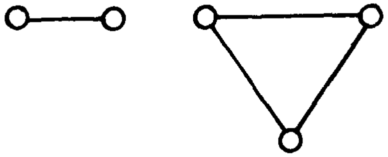
DIRECTED GRAPHS ALSO HAVE A NOTION OF CONNECTEDNESS, BUT WE WILL NOT BE CONCERNED WITH THAT.

CONNECTED GRAPH

AN UNDIRECTED GRAPH IS CONNECTED IF THERE IS A PATH BETWEEN ANY TWO VERTICES



CONNECTED GRAPH



DISCONNECTED GRAPH

INSTRUCTOR NOTES

THE NEXT SLIDE WILL SHOW WHY THEY ARE CALLED SPANNING TREES.

SPANNING TREE

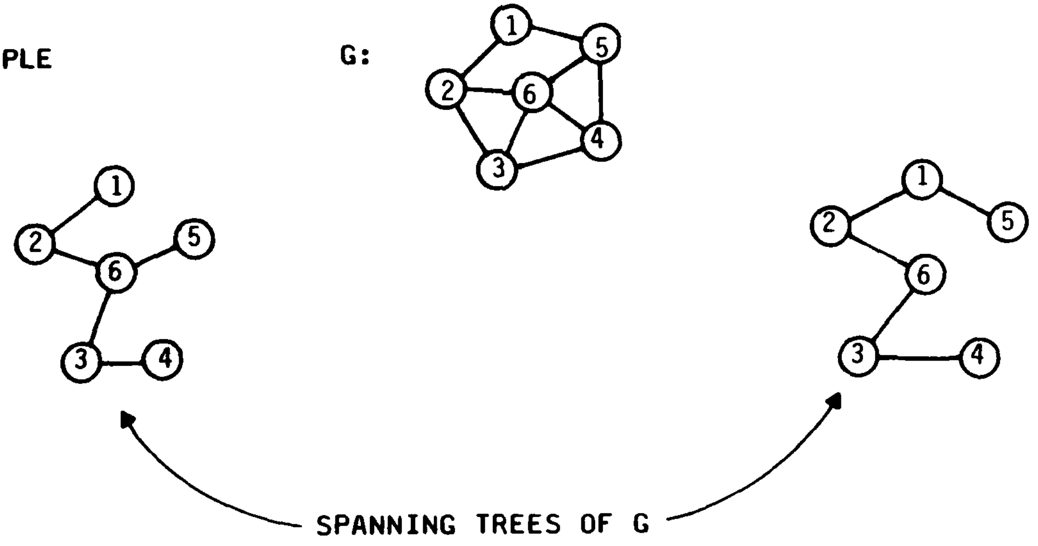
- A <u>SPANNING TREE</u> OF A CONNECTED UNDIRECTED GRAPH G IS ANOTHER CONNECTED UNDIRECTED GRAPH G' SUCH THAT

  -- THE VERTICES OF G' ARE EXACTLY THE SAME AS THE VERTICES OF G

  -- THE EDGES OF G' ARE A SUBSET OF THE EDGES OF G

  -- THERE IS NO PATH FROM A VERTEX TO ITSELF (CALLED A CYCLE)

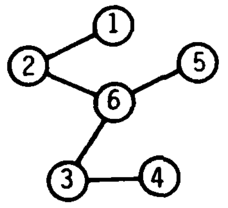
- EXAMPLE



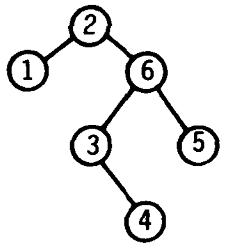
INSTRUCTOR NOTES

BY SHAKING DOWN THE SPANNING TREE WE ACTUALLY GET A TREE.

WHY SUCH GRAPHS ARE CALLED SPANNING TREES



- PICKING THE GRAPH UP BY ANY VERTEX, SAY ②, AND SHAKING IT DOWN YIELDS



INSTRUCTOR NOTES

NOTE THAT WE ARE USING THREE ABSTRACTIONS WE DEFINED EARLIER.

MINIMUM COST SPANNING TREE

- GIVEN A WEIGHTED CONNECTED UNDIRECTED GRAPH G, A MINIMUM COST SPANNING TREE IS A SPANNING TREE T SUCH THAT THE SUM OF THE COSTS OF THE EDGES OF T IS A MINIMUM FOR ALL SPANNING TREES OF G.

- USES:
  - A HIGHWAY CONSTRUCTION PLAN IS BEING DEVISED TO CONNECT SOME CITIES (VERTICES). EACH PROPOSED ROUTE (EDGE) HAS A CONSTRUCTION COST. MINIMIZE THE COST OF THE CONSTRUCTION PLAN.
  - SAME AS ABOVE EXCEPT THAT INSTEAD OF CONSTRUCTION COST, EACH ROUTE HAS AN ENVIRONMENTAL IMPACT INDEX. MINIMIZE THE ENVIRONMENTAL IMPACT OF THE CONSTRUCTION PLAN.

- SOLUTION WILL USE
  - LINKED LISTS
  - PRIORITY QUEUES
  - MERGEABLE SETS

INSTRUCTOR NOTES

THIS ALGORITHM IS SOMETIMES CALLED THE "GREEDY ALGORITHM." ITS REAL NAME IS KRUSKAL'S ALGORITHM.

ALGORITHM FOR CONSTRUCTING A MINIMUM-COST SPANNING TREE

- REPEATEDLY SELECT THE CHEAPEST UNSELECTED EDGE AND, IF IT CONNECTS TWO STILL-UNCONNECTED PARTS OF THE GRAPH, ADD THE EDGE TO THE SPANNING TREE, UNTIL ALL VERTICES IN THE SPANNING TREE ARE CONNECTED. (N-1 EDGES WILL BE NEEDED TO CONNECT N NODES.)

- WHY DOES THIS ALGORITHM ALWAYS PRODUCE THE CHEAPEST POSSIBLE SPANNING TREE?

  - AT FIRST GLANCE, IT'S OBVIOUS.

  - AS YOU THINK ABOUT IT MORE CAREFULLY, IT'S LESS OBVIOUS. (BY SELECTING THE CHEAPEST POSSIBLE EDGE AT SOME POINT, MIGHT YOU BE COMMITTING YOURSELF TO A MORE EXPENSIVE COMBINATION OF EDGES IN THE LONG RUN?)

  - IF YOU THINK ABOUT IT CAREFULLY ENOUGH TO PROVE IT, IT'S OBVIOUS ONCE AGAIN. (FOR A PROOF, SEE AHO, HOPCROFT, AND ULLMAN, SECTION 5.1.)

INSTRUCTOR NOTES

THIS SHOWS HOW WE CAN DRAW ON A LIBRARY OF GENERAL PURPOSE SOFTWARE COMPONENTS TO OBTAIN THE ABSTRACTION NEEDED TO SOLVE THE PROBLEM AT A HIGHER LEVEL.

IMPLEMENTING THE ALGORITHM

- REPRESENT THE GRAPH USING EDGE SETS
- USE MERGEABLE SETS TO KEEP TRACK OF WHICH VERTICES HAVE PATHS BETWEEN THEM. (COLLECTIONS OF VERTICES THAT HAVE PATHS BETWEEN THEM ARE CALLED CONNECTED COMPONENTS.)
    - Same_Set TELLS US THAT THERE IS A PATH BETWEEN TWO VERTICES (THEY ARE IN THE SAME CONNECTED COMPONENT).
    - WHEN ADDING AN EDGE WE ARE SAYING THAT THE VERTICES OF THE EDGE HAVE A PATH BETWEEN THEM. BUT THIS MEANS THAT THE VERTICES IN THE CONNECTED COMPONENTS OF THE TWO VERTICES HAVE A PATH BETWEEN THEM. THIS MEANS WE MUST MERGE THE TWO CONNECTED COMPONENTS, I.E., CALL Merge_Sets.
- USE A PRIORITY QUEUE TO EXTRACT EDGES IN ORDER OF INCREASING COST.

Material: Advanced Ada Topics (L305), Volume II

We would appreciate your comments on this material and would like you to complete this brief questionaire. The completed questionaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?

Yes □ No □

Comments:

3. Were there any typographical errors?

Yes □ No □

If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?

Yes □ No □

Comments:

5. General Comments:

place
stamp
here

COMMANDER
US ARMY MATERIEL COMMAND
ATTN: AMCDE-SB (OGLESBY)
5001 EISENHOWER AVENUE
ALEXANDRIA, VIRGINIA 22233

END
FILMED

4-86

DTIC